Generation of Simple Polygons and Holes in Simple Polygons with the Sweep Line Algorithm

Thesis · June 2021

DOI: 10.13140/RG.2.2.16787.40481

1 author:

Steinþór Jasonarson
University of Salzburg
2 PUBLICATIONS 10 CITATIONS
SEE PROFILE

Generation of Simple Polygons and Holes in Simple Polygons with the Sweep Line Algorithm

Master Thesis

To obtain the academic degree

Master of Science in Engineering

At the Faculty of Natural Science

of Paris Lodron University of Salzburg

Submitted by

Steinþór Jasonarson

11719070

Supervisor: Martin Held

Subject Area: Computational Geometry

Salzburg June 2021

ABSTRACT

An Inversion is a well known algorithm used to untangle intersections in a polygon, and the Sweep Line is a well known algorithm to find intersections to untangle. As inversions are applied to untangle intersections, we may introduce new intersections. It is not obvious how to combine a sweep line with inversions in an efficient way, and it is a surprise that no experimental analysis has been made on the algorithms before.

This thesis looks at analysing the practical time efficiency of the sweep line inversions that generates simple polygons and how it might affect the randomness of the generated simple polygons. Using the sweep line algorithm as a basis, we design and analyse variations to generate simple polygons on a given input point set which are on average three to six times faster than a current well known implementation. The speed increase can possibly affect the randomness of the generation process, so a criterion is developed to compare the randomness of the simple polygon generation of the variations .

This thesis also describes a new heuristic based on the Sweep Line to generate holes in a polygon. The algorithm accepts a simple polygon and any holes that are allowed to be modified as input, and uses the Sweep Line algorithm to try and find a pair of edges that can be used to generate a new hole. The algorithm either modifies an already existing hole, or modifies the enclosing polygon to generate a hole. Time complexity of a single sweep is $\mathcal{O}(n \log n)$ and memory complexity is $\mathcal{O}(n)$.

ACKNOWLEDGEMENTS

I want to thank Dr. Martin Held for the opportunity to work on the project that is the origin for the thesis, as well as thanking my colleagues in the Computational Geometry group; Peter Palfrader, Stefan de Lorenzo, Günther Eder, and Phillip Mayer for their discussions and support.

This work was supported by the Austrian Science Fund (FWF): Grant ORD53-VO.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Random simple polygons are used for verifying and testing geometric algorithms that work on polygonal data for various problems, such as algorithms that triangulate a polygon, or algorithms that minimise/maximise the area of a polygon. This makes the generation of random simple polygons, and any effort in improving time efficiency an area of interest.

In this chapter we will go over some of the different ways to generate simple polygons, and mention other notable research regarding the generation of simple polygons.

1.1 State of the Art

In 2011 Sharir [1] improved the upper bound of the number of simple polygons to $\mathcal{O}(54.543^n)$ on a given set of n points. A lower bound of $\Omega(4.64^n)$ for a specific class of input sets of n points was published by Garcia in 2000 [2].

In 2020 Nakahata et al. [3] published an algorithm that can compile connected crossing-free geometric graphs into a directed acyclic graph (DAG). It compiles crossing free spanning cycles (a.k.a. simple polygons) in $\mathcal{O}(4^n)$ of a given set of n points, and finds the smallest/largest area simple polygon in $\mathcal{O}(4.289^n)$.

1.1.1 Generation of Random Simple Polygons with a Modifiable Input Set of Points

Bouncing Vertices

Bouncing vertices is an algorithm detailed by O'Rourke and Virmani [4]. Auer describes it in his thesis [5], and Mayer modifies and extends in his thesis [6].

The Bouncing Vertices algorithm starts with an already known simple polygon P within a region R and randomly shifts the points around such that the polygon stays simple and inside R. Auer verified simplicity after each shift, while Mayer implemented a triangulation of the polygon that keeps the polygon simple before and after the shift.

No attempt at quantifying the randomness was made by O'Rourke, Auer, or Mayer, but some criterions were developed to measure randomness. Time complexity of Mayer's implementation in his FPG package is highly dependent on the number of shift operations done.

Random Orthogonal Polygons

Two algorithms to generate random orthogonal polygons was published in 2003 [7] and 2004 [8] by Tomás and Bajuelos. The algorithms start with a given number of points in a square, where each point is on a grid, then points are added with either the Inflate-cut or the Inflate-paste method. Both methods select a random cell of the grid, then split the column and row of the grid so that the selected cell is now in four pieces. A piece is randomly either cut or added (pasted) so that all cells of the polygon are still properly connected.

In 2018 Paul [9] published a paper on random simple orthogonal polygons on a given set of n points in a general position (no collinearities) that at most adds n additional points to generate an orthogonal polygon and runs in $\mathcal{O}(n \log n)$. The algorithm starts by finding the point with the highest y-coordinate v_t, and forms a vertical line ℓ_{v_t} from that point. The input point set is split into two halves based on which half-plane of ℓ_{v_t} they are in. Each point set is ordered by their y-coordinate and the process starts in the y-coordinate of the lowest point of the original point set. For each point set and each point, the intersection of the vertical line from the older point and the horizontal line of the new point is added to a polygon, which means the resulting orthogonal polygon is y-monotonic with respect to ℓ_{v_t}.

Vertex Insertion

Vertex Insertion is detailed in paper by Dailey and Whitfield [10]. It is an algorithm that generates a simple polygon by starting with a triangle of three random points. A random edge of the triangle is selected and the visibility area of both endpoints of the edge is determined. The visibility area is then triangulated and a random triangle of the triangulation is selected with a probability based on the area. A random point is then chosen from within the selected triangle. The selected point is connected with edges to the endpoints of the earlier selected edge. The algorithm runs in $\mathcal{O}(n^2 \log n)$ time.

1.1.2 Generation of Random Simple Polygons with an Immutable Input set of Points.

Uniformly Random X-Monotone Polygons

Zhu et al. [11] published a paper in 1996 that described a method that could generate uniformly at random an x-monotonic simple polygon on a given input point set $\mathcal{S}$. X-monotone simple polygons are a subclass of polygons where any x-monotone polygon can be divided into two chains; upper and lower, where both chains are monotone with respect to the x-axis. Both chains start in the left-most vertex and end in the right-most vertex.

The algorithm defines $T(k)$ as the number of polygons that contain an edge $(k-1,k)$ on the top chain, and defines $B(k)$ as the number of polygons that contain the edge $(k-1,k)$ on the bottom chain. An integer x between 1 and $T(k)+B(k)$ is randomly selected and fed into a generator that adds points to top or bottom chain given a visibility condition that is connected to the probability of picking that specific polygon.

The paper proves that the random generation is uniform which makes it an important algorithm for generating x-monotonic simple polygons.

In their paper Zhu et al. also describe a way to generate nested polygons by changing the definition and computation of the visibility, as well as describe an algorithm that generates a random convex polygon on a subset of $\mathcal{S}$.

Generation Using Convex (Onion) Layers

In 2013, Sadhu et al. [12] published an algorithm that generates a random simple polygon using convex layers. The algorithm first generates the convex hull layers of $\mathcal{S}$ in $\mathcal{O}(n^2)$ time using the Jarvis march algorithm. Starting with the outermost layer CL_1, a random edge is selected. The visibility of the selected edge to the edges of CL_2 is calculated and one of the visible edges of CL_2 is randomly selected to connect the two layers. The same is then done to connect CL_2 to CL_3, and so forth.

This algorithm generates a subset of all simple polygons on a point set. The time complexity of the process of connecting the layers is reported to be $\mathcal{O}(n \log n)$, and the space complexity is $\mathcal{O}(n)$.

Star-shaped Polygons

Auer [5] implemented three algorithms that generate star-shaped polygons, a *Quickstar* algorithm that generates a random star-shaped polygon, *Star Arrange* that is based on computing the arrangement of all lines between any two points of $\mathcal{S}$, and *Star Universe* that enumerates all star-shaped polygons on $\mathcal{S}$ and picks one uniformly at random. Star Arrange has both time and space complexity of $\mathcal{O}(n^4)$ for n points, Star Universe has time complexity of $\mathcal{O}(n^5)$ and space complexity of $\mathcal{O}(n^2 + k)$ for n points and k star shaped simple polygons. Quick star has time complexity of $\mathcal{O}(n \log n)$ and space complexity of $\mathcal{O}(n)$ for n points.

Sohler in 1999 [13] details an algorithm that generates a uniformly random star-shaped simple polygon, as well as a way to enumerate all star-shaped simple polygons so that a random one can be selected uniformly. The idea is to randomly find an intersection p of the line arrangement of the given points, then generate the faces incident to p to select a random face of the kernel subdivision to generate a star-shaped polygon without computing the whole division. The algorithm to generate a star-shaped simple polygon is reported to have time complexity $\mathcal{O}(n^2 \log n)$, and $\mathcal{O}(n)$ space complexity. The algorithm that enumerates all star-shaped polygons is reported to have time complexity of $\mathcal{O}(n^5 \log n)$ and $\mathcal{O}(n)$ space complexity.

Space Partitioning

Auer [5] implemented a recursive algorithm that partitioned the space between given points to generate a simple polygon that runs in $\mathcal{O}(n \log n)$ time. The algorithm at first selects two random points of $\mathcal{S}$: s_f, s_l, that are on the convex hull of $\mathcal{S}$. The rest of $\mathcal{S}$ is split into two subsets based on the half-planes of the supporting line of s_f and s_l. For a subset $\mathcal{S}'$, defined by two points s'_f, s'_l, a random point s' is selected. A random line ℓ through that point is generated that has s'_f, s'_l in opposite half-planes of ℓ. The line is then used to split $\mathcal{S}'$ into two subsets which go through the same recursive process. The sets are then joined together to form a polygon.

A subset of all simple polygons of a point set can be generated with the algorithm, and it has a worst case run time of $\mathcal{O}(n^2)$ as it depends on whether the sizes of the subsets of the split are roughly equal or not.

Steady Growth

Steady Growth is another algorithm presented by Auer in his thesis. Two points from $\mathcal{S}$ are selected at random to start a polygon. A point is randomly selected from $\mathcal{S}$ where the rest of the input set lies outside the convex hull of the current polygon plus the new point. The visibility of edges of the polygon to the new point is checked and a visible edge is selected at random where the point is inserted into the polygon. This algorithm will generate a subset of all simple polygons, and the time complexity is $\mathcal{O}(n^2)$.

Incremental Construction and Backtracking

First spoken of by Jefferey A. Shufelt and Hans J. Berliner in 1994 [14] and described by Auer in his thesis [5]. Further research has been done by Jiang [15] where the edge-pruning technique of the graph is improved.

The idea is to start with the complete graph of the input set. A random edge is selected to be added to the polygonal chain. The edge is checked to see if violates certain conditions that would make it impossible to complete the chain into a simple polygon. If the edge does not violate the conditions, it is added to the polygon and other edges from the graph that become unusable are removed.

In Jiang's paper they use the concept of a cycle basis of a graph instead of complete graphs and classify removable cycles as a condition to find Hamiltonian paths. Regarding time complexity they only specify that the complexity is polynomial.

Permute & Reject

An algorithm described by Auer [5] in his thesis that is simple to implement, and will return a simple polygon from a uniform random distribution. The idea is to do in a loop: generate a random polygon from a uniformly random distribution, checking if it is simple and rejecting it if it isn't.

The time complexity is what makes this a problematic algorithm, which is $\mathcal{O}(n \log n \cdot \vartheta(n))$ on n vertices, where $\vartheta(n)$ represents a function dependent on the ratio between non-simple polygons and simple polygons in the set. As a worst case example; for convex sets this ratio is $\frac{n!}{2n}$.

Untangling Intersecting Edges of a Polygon.

The origin of the ideas used to generate polygons in this thesis can be traced back to the analysis of the Traveling Salesman problem (TSP) in the 1940's by Robinson [16], and 1950's by Flood [17] and Croes [18] (among many others). At that time many approximate heuristics were developed that give α-optimum (where α is a multiple on the minimum perimeter) results of the TSP problem in a quick amount of time, followed by an untangling of any left over intersecting edges.

Flood stated that any optimal tour of the TSP would satisfy the replacement of two edges if the sum of the lengths of two new edges were shorter than the sum of the lengths of the old edges and specifically mentioned that this untangles intersections in the tour.

Croes [18] published in 1958 a paper that focused on solving the travelling salesman problem purely by untangling intersections rather than using it after another algorithm and named the specific heuristic an *Inversion*.

Shamos and Hoey [19] in 1976 proved that the sweep line algorithm can be used to detect an intersection of geometric objects, which was further extended to be able to count all intersections by Bentley and Ottman in 1979 [20].

Leeuwen and Schoone proved in 1981 that the "common procedure to remove all intersections from a tour without increasing its length, is guaranteed to terminate within polynomially many steps" [21]. They specify three types of intersections and how they can be untangled with an Inversion, or by sorting collinear points on a specific line.

Bentley details a simple polygon algorithm that uses the Sweep Line algorithm [22] and the Inversion, and gives the Inversion the name "2-Opt swap". Bentley defines: "A tour that cannot be improved by any 2-Opt swap is said to be 2-Optimal" which is possibly the origin of the "2-Opt" moniker that is used currently for the Inversion. Bentley also describes a generalisation to λ-Opt which deletes λ edges and inserts λ new edges, where a tour is λ-optimal if the search proceeds through the tour without having to make any λ insertions/deletions.

Auer in his thesis [5] details an implementation of the sweep line algorithm to find and untangle intersections of a polygon with the Inversion (which he referred to as a "2-Opt move"). The implementation uses the Bentley-Ottman sweep line variation to detect all intersections, saves the intersections in three separate lists, and randomly selects one intersection to untangle.

The Inversion and its generalisations are still being studied for the travelling salesman problem, for instance in a paper published by Brodowsky [23]. In the paper the 2-Opt heuristic is a more general version, i.e. a process is 2-Optimal if no two edges can be replaced by shorter edges, rather than only being replaced if they intersect.

This thesis will refer to the 2-Opt swap/move as an Inversion, as Croes used that moniker.

CHAPTER 2

FUNDAMENTAL DEFINITIONS

This chapter will define the notation of general concepts, as well as detail the basic design of the line-sweep algorithm used for the simple polygon generation and the hole generation. One of the main requirements is that the implementation handles collinearities, and so definitions and concepts will reflect that when necessary.

2.1 Points and Line Segments

For a given point p in $\mathbb{R}^2$, we represent its horizontal coordinate x value as p_x and its vertical coordinate y value as p_y. A lexicographical ordering of two points p and q in the plane is represented as $p < q$. The statement $p < q$ is true if for the coordinates of p and q either of the following cases are true: i) $p_x < q_x$, or ii) when $p_x = q_x$ then $p_y < q_y$.

The lexicographical ordering relative to a point p in $\mathbb{R}^2$ splits the rest of the plane into two components, a left component made out of the left half-plane combined with a vertical ray from p to $-\infty$ and a right component made out of the right half-plane combined with a vertical ray from p to ∞. Figure 2.1 shows where the left component is made up of the red area and the normal ray, and the right component is made up of the green area and the dotted ray. The point itself is a part of the right component. We say a point q is to the left of a point p if it is in the left component of p, and to the right of p if it is in the right component.

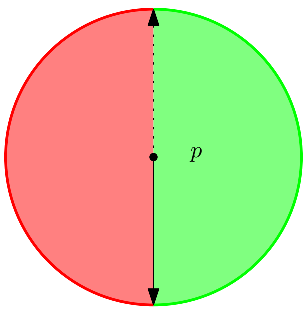

Figure 2.1: The two half-planes and the two rays from point p.

Definition 2.1.1. Area bounded by two points. Given two points p and r, and a lexicographical ordering where $p < r$, the bounded area of p and r is the intersection of the right component of p and the left component of r.

The bounded area degenerates to a vertical line segment if $p_x = r_x$, but $p_y \leq r_y$.

Assumption 1. All points in the input set are distinct.

This is verified by a lexicographical ordering of the points and checking if any adjacent points in the ordered set have the same coordinate value.

A line segment will be represented in two separate ways:

1. A lower case alphabetical symbol, such as e. When the endpoints of e are described as e_1 and e_2, it will always be true that $e_1 < e_2$.
2. Given arbitrary points a and b, the line segment with a and b as endpoints will be represented as $[a, b]$.

A lexicographical ordering of two line segments e and f is represented as $e < f$. The statement $e < f$ is true if for the endpoints of e and f either of the following cases are true: i) $e_1 < f_1$, or ii) when $e_1 = f_1$, then $e_2 < f_2$.

Two line segments are said to **intersect** if they share a point which is not an endpoint of both line segments. See Figure 2.2, where the left-most sub-figure is defined as not being an intersection. If a line segment e shares at least a point with a line ℓ, e is said to **cross** ℓ, or vice versa.

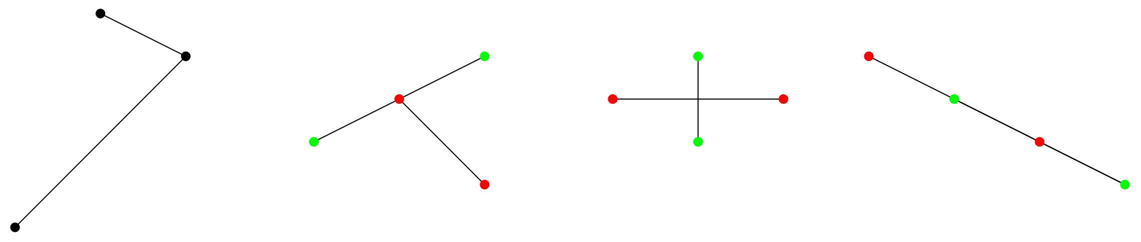

Figure 2.2: One non-intersecting pair of line segments, and three intersections of line segments.

Definition 2.1.2. Overlapping bounded areas of line segments. Given two line segments e and f and the bounded areas of their endpoints; the bounded areas of the line segments overlap, if at least an endpoint of one line segment is inside the bounded area of the other segment.

See example in Figure 2.3 where the green area plus the dotted rays is the bounded area of line segment e, and the blue area plus the dash-dot-dotted rays is the bounded area of f.

The lexicographical higher endpoint of a line segment is not a part of the bounded area, so two line segments e and f where $e_2 = f_1$ do not overlap their bounded areas.

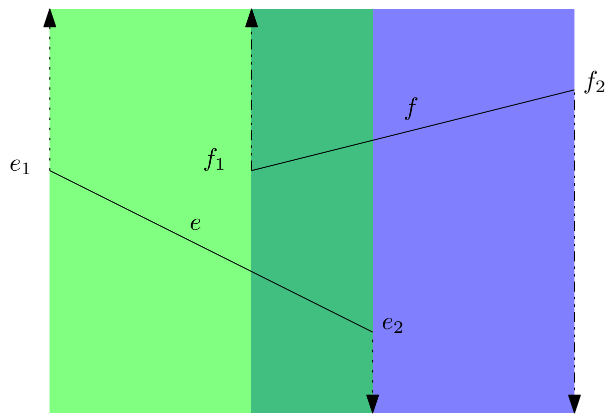

Figure 2.3: Overlapping bounded areas of line segments e and f.

Determinant

Given a line segment e and a point r, the *determinant* of the two endpoints of e and r is represented as $det(e, r)$. Given a direction from the lower lexicographical endpoint e_1 to the higher e_2, the sign of the determinant tells us in which half-plane of the supporting line of e the point r is in. To do this we need to set up the determinant in this way:

For a given line segment e with endpoints a and b, where $a < b$, and an arbitrary point r:

$$\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ r_x & r_y & 1 \end{vmatrix} \tag{2.1}$$

Figure 2.4 shows line segment e in five different rotations with its supporting line in red. If a point r is in the blue half-plane of e, then the $det(e, r)$ result is a positive value, and if r is in the green half-plane then $det(e, r)$ result is a negative value. If r is on the supporting line of e, then the resulting value is zero. The endpoints of e are shown outside of the circles to indicate which end of the line segment is which. The direction from e_1 to e_2 defines the blue half-plane as the left half-plane, and the green half-plane as the right half-plane.

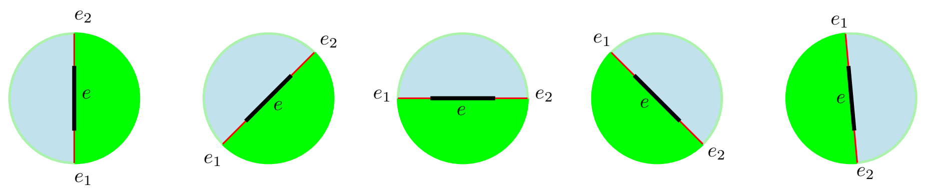

Figure 2.4: Five rotations of a line segment e, the supporting line of e is in red and the two half-planes of the supporting line of e are green and blue.

In our code the determinant is used to order line segments, see Section 2.3.2, as well as to detect intersections, see Section 3.2.

Our code uses Orient2d() algorithm from Shewchuk [24] for the determinant calculations.

2.2 Polygonal Chains

A polygonal chain is a sequence of line segments embedded in the plane that are connected at their endpoints. A polygonal chain is represented as a sequence of points in our code, and will be depicted as references to points between [ ] brackets. This depiction can be seen as an extension of the line segment representation from $[a, b]$ to $[a, b, c, d]$, where a line segment is said to be embedded in the plane between any two adjacent points in the sequence. The endpoints of the chain itself are the first and last points in the sequence.

We will use **vertex** as a reference to an endpoint of a line segment in a polygonal chain sequence, and **edge** as a reference to the embedded line segment in the plane between the points of two adjacent vertices in a polygonal chain sequence. An edge is said to be incident to the vertices that define it. Each vertex is incident to up to two edges, and each edge is incident to two vertices.

Assumption 2. All vertices in a polygonal chain sequence are distinct and are points from the given input point set.

A sequence of vertices is implemented in our code as a vector from the standard C++ library.

Definition 2.2.1. Inversion on a Chain. An Inversion on a Chain is a function that swaps the endpoints of the vertex sequence with each other, then swaps the adjacent vertices of the endpoints with each other, and continues to incrementally swap adjacent vertices on either end until all vertices have been swapped.

example: polygonal chain: $[\underbrace{a}_{v_0}, \underbrace{b}_{v_1}, \underbrace{c}_{v_2}, \underbrace{d}_{v_3}, \underbrace{e}_{v_4}, \underbrace{f}_{v_5}]$

after the inversion: $[\underbrace{f}_{v_0}, \underbrace{e}_{v_1}, \underbrace{d}_{v_2}, \underbrace{c}_{v_3}, \underbrace{b}_{v_4}, \underbrace{a}_{v_5}]$.

A vertex sequence and its inverse is the same polygonal chain when embedded in the plane, i.e. a polygonal chain is inverse-invariant, however the sequence is not. A sequence and its inverse are said to be different sequences of the same polygonal chain.

Definition 2.2.2. Cutting a Chain. Cutting a Chain is a function on a vertex sequence with n vertices using a given edge in the sequence $[v_i, v_{i+1}]$, $0 \leq i \leq n-2$.

The function takes a sequence $[v_0 ..., v_i, v_{i+1}, ..., v_{n-1}]$ and creates two sequences where one sequence is a copy of the original sequence from the left end to v_i: $[v_0, v_1, ..., v_{i-1}, v_i]$, and the other sequence is a copy of the original sequence from v_{i+1} to the right end: $[v_{i+1} ..., v_{n-1}]$.

Example: $[p_1, p_2, p_3, p_4, p_5, p_6, p_7]$ cut between $[p_3, p_4]$ gives us two sequences: $[p_1, p_2, p_3]$ and $[p_4, p_5, p_6, p_7]$. A cut between $[p_1, p_2]$ would give us a sequence: $[p_1]$ which, depending on the context, can be a valid result.

A *closed* polygonal chain is represented as a sequence of vertices and will be referred to as a **polygon**. The endpoints of the sequence are then considered adjacent in the sequence and thus connected by an edge. A polygonal chain sequence with non-adjacent endpoints is an *open* polygonal chain.

An open polygonal chain is said to **form a polygon** when the endpoints of the chain are to be connected by a line segment.

Definition 2.2.3. Joining two polygonal chains is the process of adding the vertices of one vertex sequence to the end of another vertex sequence and returning the resulting combined sequence.

Definition 2.2.4. Shifting a Polygon. Shifting a Polygon is a function that removes a vertex on one end of the sequence and adds it to the other end.

The polygon $[p_4, p_{15}, p_1, p_{10}]$ shifted once would be either $[p_{15}, p_1, p_{10}, p_4]$ if the shift is to the left, or $[p_{10}, p_4, p_{15}, p_1]$ if the shift is to the right.

A polygon when embedded in the plane is *shift-invariant*, however the sequence is not. For a polygon with n vertices, there are n different sequences of the same polygon, as we can shift it up to $n-1$ times so that the vertex v_0 of the original sequence is in another index (not accounting for inversions).

Definition 2.2.5. Cutting a Polygon. Cutting a Polygon is a function that accepts an edge $[a, b]$ in a polygon that contains it, i.e.: $[..., a, b, ...]$, shifts the vertices until $[b, ..., a]$, and returns the sequence as an open polygonal chain.

Definition 2.2.6. Splitting a Polygon is a function that cuts a polygon into a polygonal chain sequence along an edge e, and then cuts the open chain sequence along an edge f.

Definition 2.2.7. Inversion on a Polygon. An Inversion on a Polygon is the act of splitting a polygon into two chain sequences, inverting one of the chains, then joining the chains back together.

Lemma 2.2.1. An Inversion on a Polygon conserves all edges of the polygon except for possibly the two edges that were cut when splitting the polygon.

When the two sequences of the cut polygon both have more than one vertex, the inversion on a polygon will remove two edges and introduce two new edges. An inversion will have no effect on the polygon if one sequence only has a single vertex.

In an Inversion on a Polygon operation, picking either sequence to invert will result in the same polygon embedded in the plane, however the resulting sequence of the polygon is different depending on which chain was inverted and which ends were joined. To reduce the practical time cost, the shorter chain is always selected to be the one that is inverted.

2.3 The Sweep Line Algorithm

In their paper Shamos and Hoey [19] describe how to use the Sweep Line algorithm to detect an intersection of geometric objects. Bentley and Ottman [20] extended the algorithm in their paper to find all intersections.

In our implementation the Sweep Line algorithm uses an event queue which contains the vertices of an input polygon ordered lexicographically, and a line segment status object which contains a subset of the edges of the input polygon.

The metaphorical concept is that we sweep the plane with a line ℓ, starting from beyond the left-most vertex, with the sweep ending beyond the right-most vertex, where only the edges of the polygon that cross ℓ will exist in a status object of line segments. The line segments in the status object are ordered by the order in which they cross ℓ.

Event queue of points in our implementation is a lexicographical ordering of the vertices of an input polygon. When referring to a specific point in this queue we call it an **event point**.

Ascending through the event queue is processing event points in left-to-right order in incremental steps, and **descending** is processing the events points in right-to-left order.

Sweep line status of line segments (SLS). The sweep-line status in our implementation is an ordered set of line segments.

Lemma 2.3.1. Non-intersecting line segments with distinct endpoints that cross the sweep line are totally ordered by the points on the sweep-line which they cross.

This was shown by Shamos and Hoey [19]. In their paper they show that non-intersecting line segments a and b that cross the sweep line are comparable if the intersection of a with the sweep line is higher than the intersection of b with the sweep line and note that this gives a total order.

2.3.1 Processing an Event Point

In our implementation of an event queue we use a vector from the C++ standard library. Algorithm 1 is the processing of the event point queue. The SLS then contains only the line segments that cross a hypothetical vertical line that currently crosses the event point.

Algorithm 1 Processing an Event Point

```
 1: Initialise event queue $L$ with lexicographically ordered vertices of polygon $P$.
 2: Initialise the direction of the processing as a boolean $ascending$
 3: **For** an event point $l_i$ in $L$ **do**
 4:     Get the two adjacent vertices $a, b$ of $l_i$ in the polygon, where $a < b$.
 5:     Form the edges $[l_i, a]$ and $[l_i, b]$.
    When ascending $[l_i, a]$ gets processed earlier than $[l_i, b]$, vice versa for descending.
 6:     **If** $l_i < a$ and $ascending$ **then**
 7:         Insert $[l_i, a]$ into the SLS.
 8:     **Else**
 9:         Remove $[l_i, a]$ from the SLS.
10:     **If** $l_i < b$ and $ascending$ **then**
11:         Insert $[l_i, b]$ into the SLS.
12:     **Else**
13:         Remove $[l_i, b]$ from the SLS.
```

Inserting a line segment e into the SLS, or finding e for removal lets us discover which other segments are adjacent to e in the SLS. This information is used to check for intersections in the simple polygon generator, or to find candidates to generate holes in the hole generation algorithm.

2.3.2 Ordering of Line Segments

In our implementation of an SLS we use a set from the C++ standard library, which is a binary search tree, to order the line segments in the set. A set needs at least a strict weak comparator of the line segments to order them.

Lemma 2.3.2. The intersection of the bounded areas of the line segments currently in an SLS is not empty.

This is a result from how we process the event points, see Section 2.3.1.

Lemma 2.3.3. Given a non-vertical line segment e and a vertical line ℓ sweeping over e, the determinant $det(e, r)$ for an arbitrary point r on ℓ will tell us whether r is higher than the point where e crosses ℓ, or lower.

By our definition of $det(e, r)$, the half-planes of the supporting line of e are oriented such that if r is in the left half-plane r has a higher y-coordinate, and if r is in the right half-plane r has a lower y-coordinate. In the case when a line segment is collinear with the vertical sweep line, a comparison of the y-coordinates of the lower point of e and r is used.

Corollary 2.3.4. Given a set of non-intersecting line segments S with distinct endpoints and a non-empty intersection A of their bounded areas.

For any two line segments e and f in S, w.l.o.g. the order from using $det(e, f_1)$ (or $det(e, f_2)$) is a total order in A.

This follows from Lemma 2.3.2 and Lemma 2.3.3 as any vertical line sweeping e and f is at least partially inside A which means the line segments are comparable with the determinant inside A.

Definition 2.3.1. $\prec$ comparison. The symbol used for the order comparison of line segments will be $\prec$, where for given line segments e and f, if e is lower in the line sweep order than f then $e \prec f$.

Given that we are working with a polygon and using the determinant with the endpoints of the edges in our comparator, there are a few additional cases that need to be addressed.

Figure 2.5 shows the following comparison cases for two line segments e (green) and f (blue), with $e < f$:

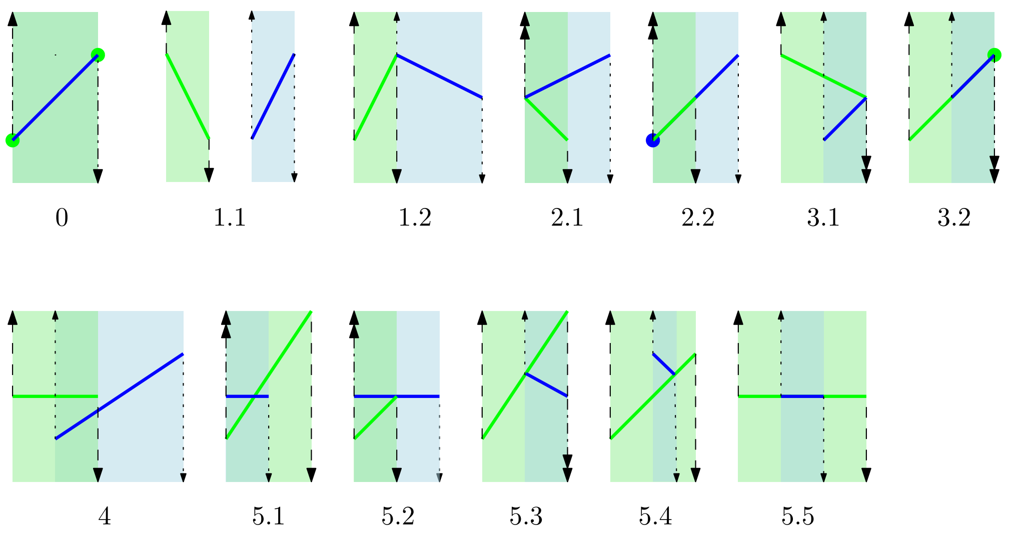

Figure 2.5: Examples of the cases when ordering line segments.

0. Should $e = f$, then $e \prec e$ returns false.
1. Line segments with non-overlapping bounded areas. This is only treated as a disaster case that should never happen given how we process the event queue. If it does come up, the lexicographical relationship of the segments is used for the comparison.
2. $e_1 = f_1$, use $det(e, f_2)$ to order the segments.
3. $e_2 = f_2$, use $det(e, f_1)$ to order the segments.
4. For non-intersecting segments with non-empty intersection of their bounded areas; use $det(e, f_1)$.
5. Intersecting segments, use $det(e, f_1)$.

The collinearities in cases 2.2, 3.2, and 5.5 in Figure 2.5 are resolved with the lexicographical relationship of the segments.

See Appendix A for a pseudo-code representation of the comparator.

2.3.3 Comparing Two Sweep Line Heuristics

There are two different implementations of the Sweep Line algorithm, one published by Shamos and Hoey [19], and the other by Bentley and Ottman [20].

Section 2.3.1 describes the basics of both heuristics. The only additional detail that is missing from the Bentley-Ottman heuristic is that if adjacent line segments in the SLS intersect, a new event point is created and added to the event queue in the appropriate place, and when processing that point, the intersecting line segments are swapped in the SLS to keep the adjacency of segments in the SLS correct.

The Bentley-Ottman version has time complexity of $\mathcal{O}(n \log n + k \log n)$ where n are the number of vertices and k are the number of processed intersection events. Figure 3.7 in Section 3.4 represents an example where the number of intersections is $\mathcal{O}(n^2)$. This means a theoretical upper limit of Bentley-Ottman is $\mathcal{O}(n^2 \log n)$.

One of the problems with Bentley-Ottman is that the new event point has to be calculated exactly if we want to deal with collinearities, otherwise we risk that the insertion of the event point is in the wrong place, and the wrong segments are adjacent in the SLS.

Our implementation in regard to generating simple polygons will look at untangling an intersection as soon as it is found. This removes the problem of having to insert new events exactly into the event queue. Using the simpler Shamos and Hoey implementation the theoretical time complexity of a single sweep is $\mathcal{O}(n \log n)$ for n line segments.

Given a sweep over intersecting line segments in the SLS, the line segments in the SLS are possibly not ordered correctly at all times based on a true line ordering in the plane.

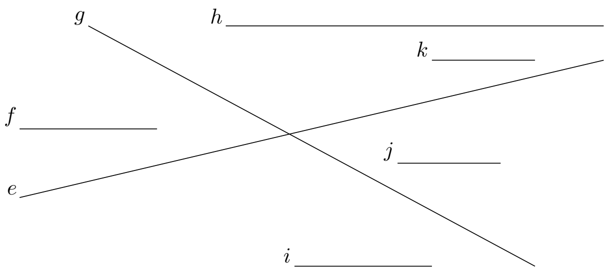

Figure 2.6: Example of line segments that are to be swept over with a sweep-line.

Example (see Figure 2.6): Assume an ascending direction. At the event point where we insert j into the SLS, the SLS is $\{i, e, g, h\}$. Assume j is compared first with e, $j \prec e$ so it becomes $\{i, j, e, g, h\}$. At the event point where we remove j, the SLS is $\{j, e, g, k, h\}$. Assume j is first compared with g, $g \prec j$ so the binary tree algorithm will look for j above g. As the SLS inserted j below e, we will not find j above g, and so we cannot find the edge to remove it.

Any implementation using the Shamos and Hoey version to untangle intersections must then sweep through the polygon without finding a single intersection, i.e. we do a new sweep from the beginning after finishing the current sweep if an intersection is found.

For the hole generation algorithm, as the assumption is that the input polygon is simple, the simpler version of the line-sweep is acceptable.

CHAPTER 3

SIMPLE POLYGON GENERATION ALGORITHM

In this chapter we will detail the implementation of the Line Sweep algorithm. The input is a set of points on which a random polygon is generated with the Shuffle algorithm [25]. The Sweep Line algorithm (see Section 2.3) is used to discover adjacent edges in the SLS, the edges are then checked for intersections, and if an intersection is found it is immediately untangled. The idea posited is that immediately untangling the intersection will lead to a reduction in the practical time complexity compared to counting all intersections and randomly selecting one to untangle.

Untangling an intersection as soon as it is detected opens up opportunities to investigate into different variations based on the decision how to continue after finding and untangling an intersection and see if the different variations will affect the practical time efficiency, or the randomness of the generation of simple polygons.

This chapter will go over the general process of detecting intersections, describe the variations we will analyse, then detail the intersection check and the untangling process in regard to how they handle collinearities.

3.1 Processing a Sweep Line Event

Algorithm 2 is the general procedure for an event point l_i in the event queue with incident edges e and f, where e is processed before f. The variations are built around deciding what to do after untangling an intersection which happens after an event point is processed. This can be to ascend to the next event point l_{i+1}, to descend to the next event point l_{i-1}, to repeat the event point or to reset the SLS and start again from l_0, or from a random point l_r.

Algorithm 2 Processing edges connected to an event point

```
1: Edges $e$ and $f$ are incident to an event point $j$ with $e$ processed before $f$.
2: If $e$ is collinear with $f$ then
3:     Untangle intersection between $e$ and $f$.
4:     Stop further process of the event point.
5: If $e$ should be removed from the SLS then
6:     Get adjacent edges $a$ and $b$ of $e$ in the SLS.
7:     Remove $e$.
8:     If $a$ and $b$ intersect then
9:         Untangle the intersection and remove $a, b$ from the SLS.
10:        Stop further process of the event point.
11: If $e$ should be inserted into the SLS then
12:     Insert $e$ into the SLS.
13:     Get adjacent edges $a$ and $b$ of $e$ in the SLS.
14:     If $a$ or $b$ intersect with $e$ then
15:         Remove the intersecting edges from the SLS.
16:         Untangle the intersection.
17:         Stop further process of the event point.
18: If $f$ should be removed from the SLS then
19:     Get adjacent edges $a$ and $b$ of $f$ in the SLS.
20:     Remove $f$.
21:     If $a$ and $b$ intersect then
22:         Untangle the intersection and remove $a, b$ from the SLS.
23:         Stop further process of the event point.
24: If $f$ should be inserted into the SLS then
25:     Insert $f$ into the SLS.
26:     Get adjacent edges $a$ and $b$ of $f$ in the SLS.
27:     If $a$ or $b$ intersect with $f$ then
28:         Remove the intersecting edges from the SLS.
29:         Untangle the intersection.
```

3.2 Line Segment Intersection Check

When processing an event point an intersection check is necessary in two cases: i) at the insertion of an edge e into the SLS, where e is checked against the adjacent edges a or b in the SLS, or ii) at the removal of an edge e, where a is checked against b.

The check accepts two edges e and f as input and should discern between four cases, see Figure 3.1 as examples of the three latter cases:

1. Edges do not intersect.
2. Edges intersect and are not collinear.
3. Edges share an endpoint, overlap their bounded areas, and are collinear.
4. Edges have distinct endpoints, overlap their bounded areas, and are collinear.

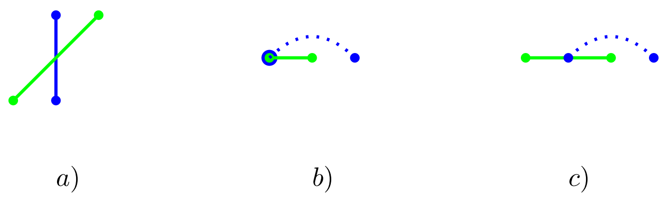

Figure 3.1: Intersecting line segments. Blue line segment is bent for clarity.

Using the determinant we can find intersections by doing four determinant comparisons. For each line segment e and f, a determinant check is done with the two endpoints of the other line segment, i.e. for e: $det(e, f_1), det(e, f_2)$, and for f: $det(f, e_1)$, $det(f, e_2)$.

To check if two line segments e and f overlap, we can sort the four endpoints. If they do not overlap, the result of a sort should either be $\{e_1, e_2, f_1, f_2\}$ or $\{f_1, f_2, e_1, e_2\}$.

See Algorithm 3 for a pseudocode of the intersection check.

Algorithm 3 Checking for Intersections

```
1:  Get the four determinants of the two edges $e$ and $f$.
2:  If $e$ and $f$ share an endpoint and overlap their bounded areas then
3:      If a determinant between the three distinct endpoints is zero then
4:          Return a three point collinearity.
5:  Else If $e$ and $f$ have four distinct endpoints and overlap their bounded areas then
6:      If all four determinants are zero then
7:          Return a four point collinearity.
8:      If a determinant for one edge is zero then
9:          If the signs of the two determinants of the other edge are opposites then
10:             Return an intersection.
11:     If the signs of the two determinants for each edge are opposites then
12:         Return an intersection.
13: Return no intersection.
```

3.3 Untangling Line Segment Intersections in a Polygon

Leeuwen and Schoone [21] published a paper in 1980 about the types of intersections and how to untangle them. Their paper describes three intersections: i) intersection between two line segments with distinct endpoints (Figure 3.2), ii) intersection between two line segments with distinct endpoints and one endpoint of one line segment collinear with the other line segment (Figure 3.3), and iii) intersection between collinear line segments (Figure 3.4 and 3.5).

In this section we will analyse these three intersections in regard to the invariant that is used to guarantee that a random polygon can be made simple by untangling intersections, and describe the untangling process in each case.

Case I: Non-Collinear Intersecting Line Segments

Theorem 3.3.1. Using the Inversion on a Polygon function along intersecting non-collinear line segments with distinct endpoints replaces the intersecting edges with shorter edges.

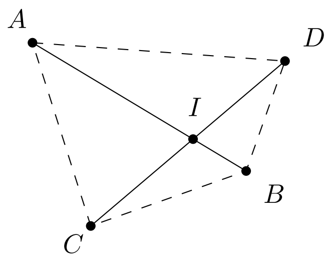

Figure 3.2: Example line segment intersection and its quadrilateral.

Figure 3.2 is a visual aid for the following proof.

Proof. Given a polygon P, four distinct non-collinear points A, B, C, D and the line segments $[A, B]$ and $[C, D]$ that intersect.

The result from using the Inversion on a Polygon are either the edges $[A, C]$ and $[B, D]$ or $[A, D]$ and $[B, C]$.

We define the point I for the intersection, and construct the simple polygon $[A, C, B, D]$ which represents a quadrilateral. The quadrilateral can be split into four triangles: $[A, C, I]$, $[C, B, I]$, $[B, D, I]$ and $[D, A, I]$.

Splitting the lengths of intersecting line segments into its elements gives:

$$\begin{aligned} |A, B| &= |A, I| + |B, I| \\ |C, D| &= |C, I| + |D, I| \end{aligned} \tag{3.1}$$

The sum of the line segments as the sum of the elements:

$$|A, B| + |C, D| = |A, I| + |B, I| + |C, I| + |D, I| \tag{3.2}$$

The triangle inequality theorem says that:

$$\begin{aligned} |A, C| &< |A, I| + |C, I| \\ |B, C| &< |B, I| + |C, I| \\ |B, D| &< |B, I| + |C, I| \\ |A, D| &< |A, I| + |C, I| \end{aligned} \tag{3.3}$$

which means:

$$\begin{aligned} |A, C| + |B, D| < |A, I| + |C, I| + |B, I| + |D, I| = |A, B| + |C, D| \\ |B, C| + |A, D| < |C, I| + |B, I| + |A, I| + |D, I| = |A, B| + |C, D| \end{aligned} \tag{3.4}$$

□

Case II: Intersecting Line Segments with One Distinct Collinear Endpoint

Theorem 3.3.2. Given an intersection between two line segments with distinct endpoints, and with one endpoint in the intersection of the two line segments.

An inversion on a polygon along intersecting line segments replaces the intersecting line segments with shorter line segments.

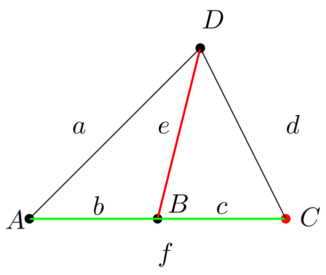

Figure 3.3: The triangle area of a three point collinearity of four distinct points.

Figure 3.3 is a visual aid for the following proof.

Proof. W.l.o.g. for four points A, B, C, D, where only points A, B, C are collinear, with point B between points A and C on their supporting line, we have the line segments $e = [B, D]$ and $f = [A, C]$.

By definition of an inversion on a polygon, the new line segments from an inversion are either $a = [A, D]$ and $c = [B, C]$, or $b = [A, B]$ and $d = [C, D]$.

We know the triangle inequalities: $a < e + b$ and $d < e + c$

$$a < e + b \iff a + c < e + b + c \iff a + c < e + f \tag{3.5}$$

$$d < e + c \iff d + b < e + c + b \iff d + b < e + f \tag{3.6}$$

□

Case III: Intersections Between Collinear Line Segments

The inversion on a polygon cannot reliably handle intersections of collinear line segments.

Example 1: three point collinearity with a shared endpoint.

Given a polygon $[a, b, d, c]$ where a, b, c are collinear, and $a < b < c$, see Figure 3.4 where the dotted line segment $[a, c]$ is bent for clarity. Line segments $[a, b]$ and $[a, c]$ intersect. An inversion on the polygon along those edges would mean inverting one of two chains: $[a]$ and $[c, d, b]$, where inverting $[a]$ has no effect, and inverting $[c, d, b]$ has no effect.

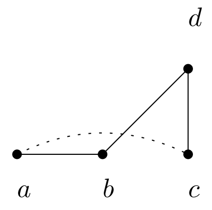

Figure 3.4: Polygon with a three point collinear intersection, one collinear edge bent for clarity.

Example 2: four point collinearity.

Given a polygon $[a, e, b, c, f, d]$ where a, b, c, d are collinear and $a < b < c < d$, see Figure 3.5. Line segments $[a, d]$ and $[b, c]$ intersect. Cutting the polygon along those edges would result in two chains: $[a, e, b]$ and $[c, f, d]$, where i) inverting $[a, e, b]$ results in $[b, e, a, c, f, d]$ where $[b, d]$ now intersects $[a, c]$, or ii) inverting $[c, f, d]$ results in $[a, e, b, d, f, c]$ where $[b, d]$ now intersects $[a, c]$.

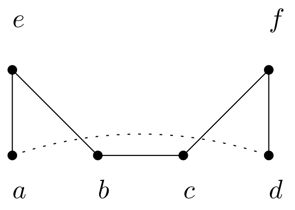

Figure 3.5: Polygon with a four point collinear intersection, one collinear edge bent for clarity.

The solution is to sort the collinear points.

3.3.1 The Collinear Sort

We discern between two collinear cases:

1. Three point collinearity (3PC), where the intersecting collinear line segments share a common endpoint.
2. Four point collinearity (4PC), where the intersecting collinear line segments have distinct endpoints.

The difference lies in how many edge connections there are to the rest of the polygon. A 3PC intersection defines a single subset chain that connects to the rest of the polygon from its endpoints, and the 4PC intersection defines two possible subset chains that have four connecting edges to the rest of the polygon.

Problem: Adjacent Collinear Vertices of Collinear Edges in the Polygon

One question to ask is whether it is enough to sort just the points of the detected collinear intersection. A 3PC could sort only the three endpoints, and the 4PC could sort the four endpoints. However, there is a problem with collinear edges when the collinear intersecting line segments that were found are also collinear with other adjacent vertices in the polygon.

operation	**polygon representation**
original	$[..., a, 0, 1, 2, 4, b, ..., c, \underline{5, 3, 8}, d, ...]$
3PC	$[..., a, \underline{0, 1, 2, 4, b, ..., c, 3, 5, 8}, d, ...]$
inversion	$[..., a, 8, \underline{5, 3}, c, ..., b, \underline{4, 2}, 1, 0, d, ...]$
4PC	$[..., a, \underline{8, 2, 3}, c, ..., b, 4, 5, 1, 0, d, ...]$
3PC	$[..., a, \underline{2, 3, 8, c, ..., b, 4, 5, 1, 0}, d, ...]$
inversion	$[..., a, 0, \underline{1, 5}, 4, b, ..., c, 8, \underline{3, 2}, d, ...]$
4PC	$[..., a, 0, 1, 2, 4, b, ..., c, \underline{8, 3, 5}, d, ...]$
inversion	$[..., a, 0, 1, 2, 4, b, ..., c, 5, 3, 8, d, ...]$

Figure 3.6: an infinite loop because of collinearity of adjacent vertices.

Example: assume we start with a polygon $[..., a, 0, 1, 2, 4, b, ..., c, 5, 3, 8, d, ...]$ where the lexicographical sorting of the identified points is $\{0, 1, 2, 3, 4, 5, 8, a, b, c, d\}$, and where we know the numerical points are collinear, see Figure 3.6.

In Figure 3.6 the first row is the original representation, and underlined is the chain which the operation detailed in the second row applies to, i.e. **3PC** is applied to $[5, 3, 8]$. The resulting representation is in the second row, and the underlined chain in that representation is the operation in the next row, i.e. **inversion**, and so on down the rows.

Given that we end up with the original polygon representation, we have an infinite loop.

To stop this from happening, we add vertices of the adjacent chains to the collinear edges that are collinear as well. Three point collinearity has two adjacent chains that need to be checked for adjacent collinear vertices, and four point collinearity has four adjacent chains that need to be checked for adjacent collinear vertices.

Algorithm 4 describes the sorting process for a 3PC intersection, and Algorithm 5 describes the sorting process for a 4PC intersection.

It is a design decision to keep the number of vertices in each subchain of the 4PC collinearity the same as it means the rest of the polygon does not need to be manipulated. Other strategies would be to randomise how many of the points are in each subchain, or to randomise whether lexicographic order of the chains is put in left-to-right or right-to-left order into the representation, or both.

The collinear sort guarantees that chains in the polygon that are collinear and intersect are shortened so that they no longer intersect.

Algorithm 4 Three Point Collinearity

```
1:  Input: two edges with three distinct endpoints $a, b, c$, that form a chain $[a, b, c]$ in polygon
    $P$.
2:  Add $a, b, c$ to a set of points $T$.
3:  **For** adjacent point $i$ in $P$ to the left of the chain **do**
4:      **If** i is not collinear with $[a, c]$ **then**
5:          Stop the loop.
6:      Add $i$ to $T$ and decrement $i$.
7:  **For** adjacent point $i$ in $P$ to the right of the chain **do**
8:      **If** i is not collinear with $[a, c]$ **then**
9:          Stop the loop.
10:     Add $i$ to $T$ and increment $i$.
11: Sort the points in $T$ lexicographically.
12: Insert the points in lexicographical order back into $P$.
```

Algorithm 5 Four Point Collinearity

```
1: Input: two edges $[a, b], [c, d]$ of polygon $P$ with four distinct endpoints.
2: Add $a, b, c, d$ to a set of points $T$.
3: Initialise $v_{left1}$ as $a$, $v_{right1}$ as $b$, $v_{left2}$ as $c$, $v_{right2}$ as $d$.
4: For adjacent vertex $i$ to the left of $a$ do
5:     If $i$ is not collinear with $[a, b]$ then
6:         Stop the loop.
7:     If $i \in \{v_{left2}, v_{right2}\}$ then
8:         Stop the loop.
9:     Add $i$ to $T$, update $v_{left1}$ and decrement $i$.
10: For adjacent vertex $i$ to the right of $b$ do
11:     If $i$ is not collinear with $[a, b]$ then
12:         Stop the loop.
13:     If $i \in \{v_{left2}, v_{right2}\}$ then
14:         Stop the loop.
15:     Add $i$ to $T$, update $v_{right1}$ and increment $i$
16: For adjacent vertices $i$ to the left of $c$ do
17:     If $i$ is not collinear with $[c, d]$ then
18:         Stop the loop.
19:     If $i \in \{v_{left1}, v_{right1}\}$ then
20:         Stop the loop
21:     Add $i$ to $T$, update $v_{left2}$ and decrement $i$
22: For adjacent vertices $i$ to the right of $d$ do
23:     If $i$ is not collinear with $[c, d]$ then
24:         Stop the loop.
25:     If $i \in \{v_{left1}, v_{right1}\}$ then
26:         Stop the loop.
27:     Add $i$ to $T$ and update $v_{right2}$.
28: Sort $T$ lexicographically.
29: Insert the points in $T$ back into $P$ in order between $[v_{left1}, ..., v_{right1}]$ and $[v_{left2}, ..., v_{right2}]$.
```

3.4 Variations on the Sweep Line Theme

The starting random polygon contains an unknown number of intersections. The upper bound is $\mathcal{O}(n^2)$ for a polygon with n points, see an example in Figure 3.7 of an input point set with only one simple polygon.

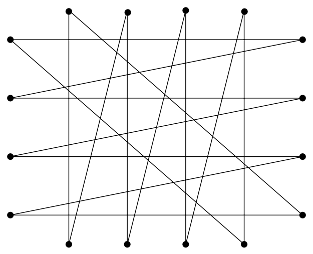

Figure 3.7: A polygon with $\mathcal{O}(n^2)$ intersections.

The untangling of an intersection removes an unknown amount of intersections and introduces an unknown amount of intersections. Figure 3.8 shows that a single untangling can theoretically introduce $\mathcal{O}(n)$ new intersections for a polygon with n edges.

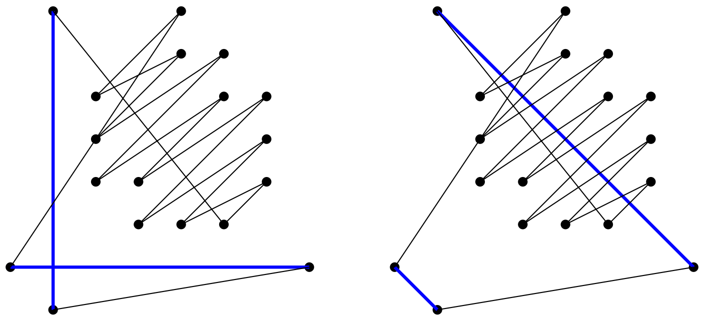

Figure 3.8: Example of an inversion on a polygon introducing more than one new intersections.

The invariant is that the perimeter gets reduced (see Section 3.3) so eventually we are guaranteed to end up with edges that do not intersect, but it is not obvious whether an application that improves the time complexity of the heuristic could affect the randomness in the distribution of the generated polygons.

Manipulating the Starting Point

Given that we untangle an intersecting as soon as it is found, only intersections between the adjacent edges currently in the SLS will be discoverable.

Looking at the example in Figure 3.7, each edge has many intersecting edges. Figure 3.9 shows a traversal of the event queue for the polygon in Figure 3.7. The right sub-figure shows a descent from point 11 to a discovered intersection in point 10. It is an example of how the starting position and the direction has a clear impact in which intersection is first detected. The left sub-figure shows an ascent from point 2 in the event point queue that finds an intersection in point 4. It's clear that it would not matter if we started in point 1 to point 3, the same intersection between $[3, 16]$ and $[4, 11]$ would be found, which means the order of adjacency in the SLS matters.

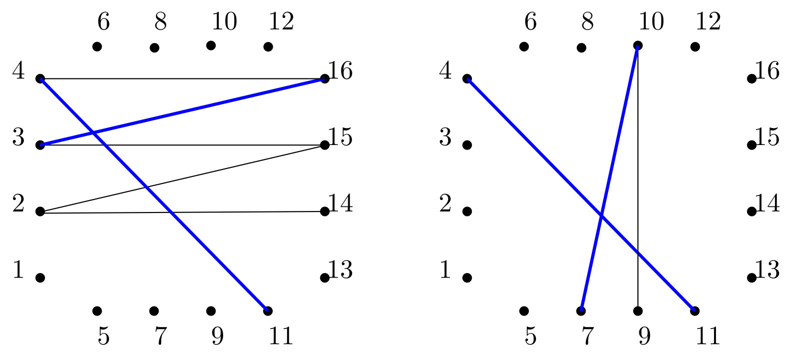

Figure 3.9: Points are lexicographically ordered, intersecting edges depicted in blue. Left sub-figure shows the status of the SLS after starting in point 2 and ascending until an intersection is discovered. Right subfigure shows the status of the SLS after starting in point 11 and descending until an intersection is discovered.

Figure 3.9 also suggests that an incomplete SLS would increase the randomness of which intersection is found. If for instance in the left sub-figure the edges connected to point 3 hadn't been added, the edge $[4, 11]$ would intersect $[2, 15]$ instead.

The Variations

In the text below, l_0 is the first event point in the event queue.

1. **Reset to zero**: this variation is only for comparison and benchmarking. It begins in l_0 and ascends through the event points. After finding and untangling an intersection, it clears the SLS and restarts in l_0.

2. **Reverse**: this variation begins in l_0 and ascends through the event points. After finding and untangling an intersection, it descends to the event point index of the lexicographically lowest endpoint of the intersection. Then at the lowest event point it ascends again. If more intersections are found while descending, it untangles them, updates the lowest point it is descending towards (if necessary) and continues descending.

3. **Repeat index**: this variation begins in l_0 and ascends through the event point index. After finding and untangling an intersection, it repeats processing the same event point until no intersections are found with adjacent edges in the SLS. Then it ascends to the next event point with the SLS in its current status.

4. **Continue**: this variation begins in l_0 and ascends through the event points. After finding and untangling an intersection, it removes the edges from the SLS, then continues ascending to the next event point without inserting any new edges into the SLS.
5. **Random**: This variation begins in a random index of the event points. It then randomly ascends/descends through the event index. After untangling an intersection it clears the SLS and restarts in a new random index.

3.4.1 Reset to Zero Variation

This variation is only interesting for analysis purposes. After untangling a found intersection, it clears the SLS and starts again from l_0, which means the lower half-plane of the sweep-line should always be empty of intersections.

See Figure 3.10 for an example of a polygon and the first five found intersections.

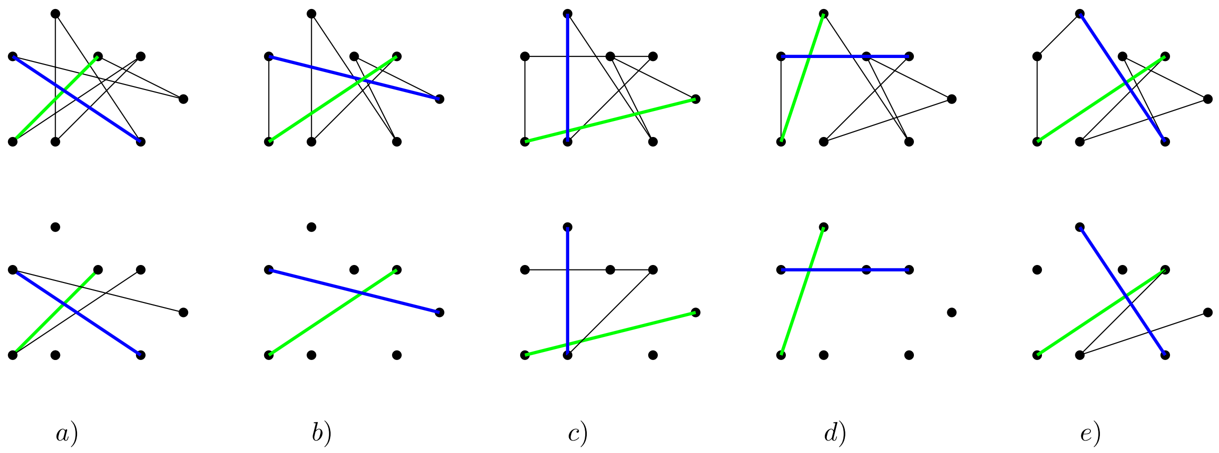

Figure 3.10: Reset to zero/Reverse: Upper row shows the polygon and the discovered intersection between the green and blue edge. The lower row shows the edges currently in the SLS when the intersection is discovered.

Expected result: i) the variation has to go through all the event points for each intersection, so this should be the slowest variation, and ii) the lower half-plane of the sweep-line has to be clear of intersections before continuing, that should affect the distribution of the random generation of polygons.

3.4.2 Reverse Variation

The variation starts in l_0 and ascends through the event point sequence. Once an intersection is found, the lowest event point index of the four vertices involved in the intersection is saved and the sweep line descends down to that lowest index. If other intersections are found along the way, the intersections are untangled and the saved index is updated if the lowest index of the other intersections is lower than the saved index. At the saved lowest index the sweep line will ascend through the event point sequence from that index.

The Reverse variation behaves similar to the Reset to zero variation in regard to never allowing intersections in the lower half-plane of the sweep-line. Because of the possibility of untangling intersections during the descent, the generated simple polygon does not necessarily have to be the exact same as the polygon generated from the Reset to zero variation.

The first five events in Figure 3.10 are the same for Reverse as for Reset to zero.

Expected result: i) the variation goes through the minimum amount of points necessary to catch all intersections, this should then be a fast variation, and ii) same as with Reset to zero, the effect on the distribution of randomly generated polygons should be high as it always backs up to the lowest clear event point before continuing, so the left half-plane of the sweep line is always clear.

One additional test was done on the Reverse variation, which was to save the SLS at the event k an intersection is found, and if reversing to a lower event point found no other intersection, to load the SLS for event point k and continue from there. From preliminary experiments, this was found to always be slower, so it wasn't analysed in-depth.

The result suggests that the sum of linear time spent saving the SLS set and throwing it away when it finds an intersection, which is relative to the number of intersections introduced by an untangling, plus the sum of linear time spent restoring the SLS if no intersection is found, is more than the time saved from not having to traverse through the line-sweep back to event point k in $\mathcal{O}(n \log n)$ time.

3.4.3 Repeat Index Variation

This variation starts in l_0 and ascends through the event queue. When an intersection is found, the edges are removed from the SLS, the intersection is untangled, and then the same event point is processed again. Only after no intersection is found in the event point does the sweep move onto the next event point. It does not re-insert any changes to edges connecting to any other event points than the current event point.

See Figure 3.11 for an example of a polygon and the first five found intersections.

Expected result: i) as the variation solves more than one intersection per sweep, it should be faster than the Reset to zero variation, and ii) because of the more random element of which edges are inside the SLS, the distribution of generated polygons for Repeat Index should be

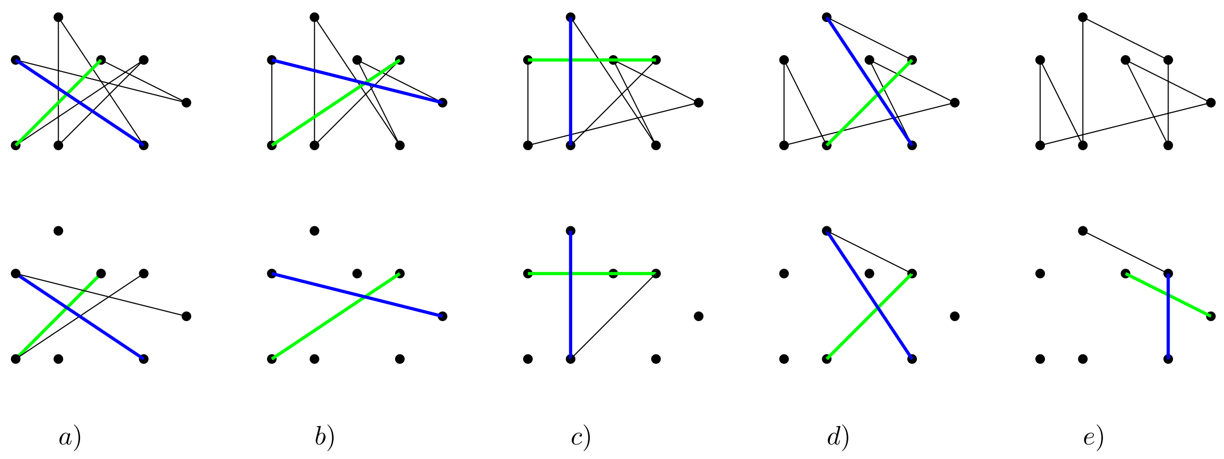
Figure 3.11: Repeat index: Upper row shows the polygon and the intersecting edges, lower row shows the edges in the SLS. The green and blue edges are the found intersection.

less biased than Reset to zero.

3.4.4 Continue Variation

This variation starts in l_0 and ascends through the event queue. Once an intersection is found, the edges of the intersection are removed from the SLS, the intersection is untangled and the next ascending event point is then processed.

See Figure 3.12 for an example of a polygon and the first five intersection events.

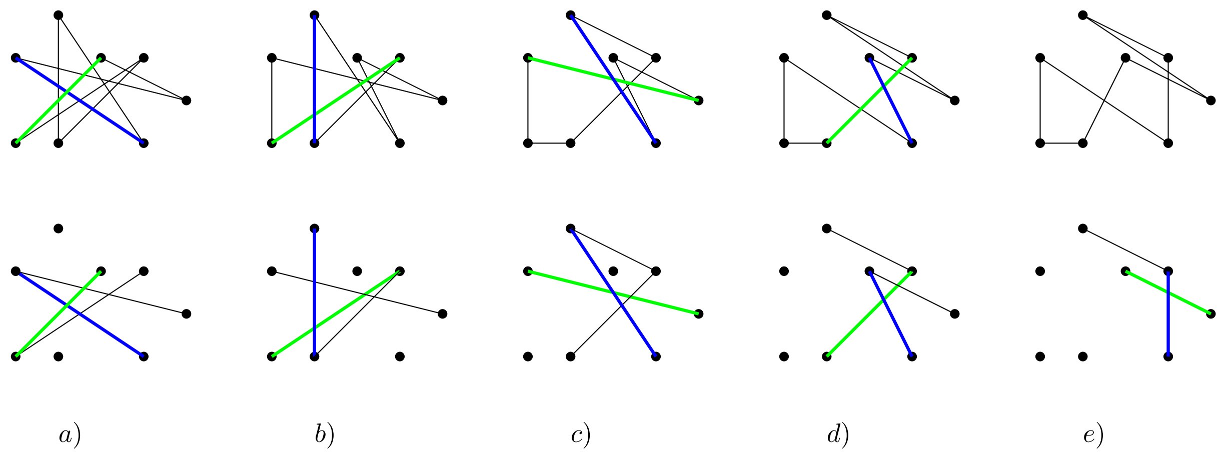
Figure 3.12: Continue: upper row shows the polygon, lower row the edges in the SLS.

Expected result: i) the variation should be faster than the Reset to zero as it can untangle more than one intersection in each sweep, and ii) The status of the SLS is not up to date, so which intersection gets untangled in each pass is more random. The variation should then be less biased in the distribution of generated polygons than the Reset to zero variation.

3.4.5 Random Variation

This variation starts in a random event point, and randomly selects to ascend or descend through the event points. If it finds an intersection, it clears the SLS and restarts in a new random event point with a new random direction.

To save time, the random variation keeps a tab on the lowest and highest event indices that are clear. To clarify: when a loop starts at a random index i, and reaches the highest/lowest point of the event point sequence, that means there are no intersections from i to the end. This means when we sweep in the same direction, we do not need to pick a random event point that is in this range $\{i, ..., end\}$ which was cleared to the end. When an intersection is found, the event indices of the endpoints are checked against these lower/ higher boundaries, and if necessary, the boundaries are changed. This reduces the amount of points that need to be checked that we already know cannot intersect.

When the higher bound - lower bound is less than three points, we switch to the Continue variation.

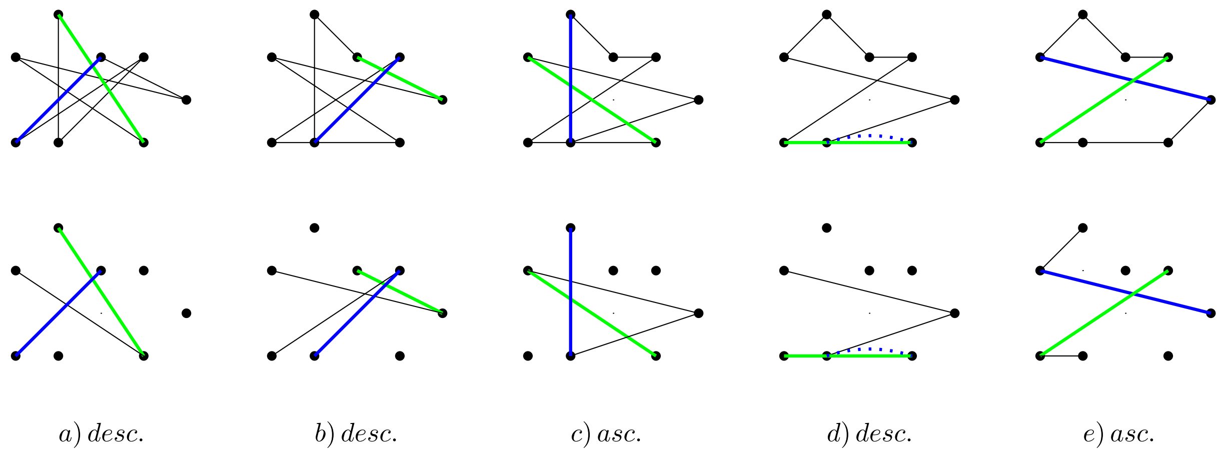

Figure 3.13: Random: the upper row shows the polygon, lower row the edges in the SLS. Blue edge in $d)$ is bent for clarity.

Figure 3.13 shows an example polygon and five intersections, after selecting a random starting index and a random direction. When ascending, the lower endpoint of the blue edge is the event point where the intersection is discovered, when descending the higher endpoint of the blue edge is the event point where the intersection is discovered.

Expected result: i) because we restart the process after every intersection, this should be one of the slower variations. The speed-up of using the boundary points should help, so the variation should be faster than Reset to zero, and ii) this variation should show less of an effect in the distribution of generated polygons than Reset to zero.

CHAPTER 4

HOLE GENERATION

This chapter will go over a heuristic that can generate holes in a polygon, sometimes referred to as nested polygons, with the Sweep Line algorithm.

In literature, hole generation algorithms have been an added part of a simple polygon generation, example is the generation of nested polygons in Zhu's paper [11], or Mayer's generation of holes [6]. The heuristic in this thesis is not connected to a simple polygon generator and so can be used on any generated simple polygon from any other algorithm.

The general idea is to find and remove chains from a simple polygon on a point set, or given a hole polygon, split it into two hole polygons to form more holes. The simple property will be assumed when talking about polygons/holes in this chapter, as both the input and the output must be simple polygons.

Assumption 3. The input set of polygons are i) simple polygons, ii) any polygon beyond the first one is nested in the first polygon, and iii) all given polygons are allowed to be modified to generate additional holes.

Given an input of a single polygon, there are degenerate cases of simple polygons which can not be modified to generate holes with this heuristic; the simplest example is a convex polygon.

Theorem 4.0.1. The Jordan Curve theorem
The complement $R^2 \setminus P$ of any simple polygon P in the plane has exactly two connected components. One of these components is bounded and the other one is unbounded, and the polygon P is the boundary of each component.

It is common to call the bounded component the **interior** of P, and the unbounded component the **exterior** of P.

Given an edge in a polygon P, the interior of the polygon will be locally to one side of the edge, we call this side the **interior side** of an edge. Which side of an edge is interior is found by setting the orientation of the polygon representation so the interior is always on the left side of an edge given a direction from v_i to v_{i+1} in the polygonal sequence.

Definition 4.0.1. Hole. A simple polygon Q is a hole of another simple polygon P if Q lies completely in the interior of P. We say the polygon Q is nested in P.

Definition 4.0.2. Diagonal. A diagonal is a line segment with non-adjacent vertices of a polygon as endpoints. If the diagonal lies completely inside the polygon (no intersections with any edges of the polygon) it is called an interior diagonal.

A point a on a line segment e is said to be **visible** from a point b on another line segment f if the line segment $[a, b]$ is interior to the polygon, and no other edges intersect $[a, b]$.

4.1 Splitting a Nested Polygon

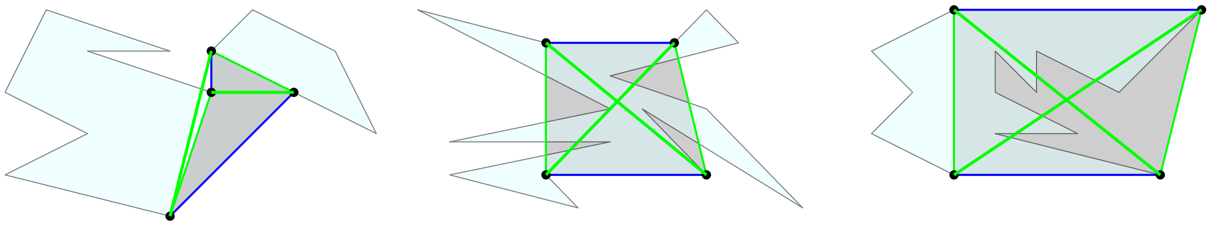
Figure 4.1: The two blue edges are the edges being considered for splitting the polygon. The green edges are diagonals.

Our strategy to generate a new hole from an existing hole Q will be to use the Sweep Line algorithm on Q to find a pair of adjacent edges in the SLS that have at least three interior diagonals between the endpoints, then split Q into two polygons along those two edges.

Figure 4.1 shows three examples of pairs of edges where only the left sub-figure has at least three non-intersecting interior diagonals.

Definition 4.1.1. Suitable pair of edges. A suitable pair of edges is a pair of edges with distinct endpoints and at least three interior diagonals between the endpoints of the edges.

For a nested polygon Q, by definition of Splitting a Polygon, using it on Q with a suitable pair of edges results in two separate simple polygons, which means a hole is split into two holes, see example in Figure 4.2.

There are two degenerate cases in regard to splitting holes, i) holes in P that do not have at minimum six vertices can not be split into two holes, and ii) holes with no suitable pairs of edges can not be split into two holes.

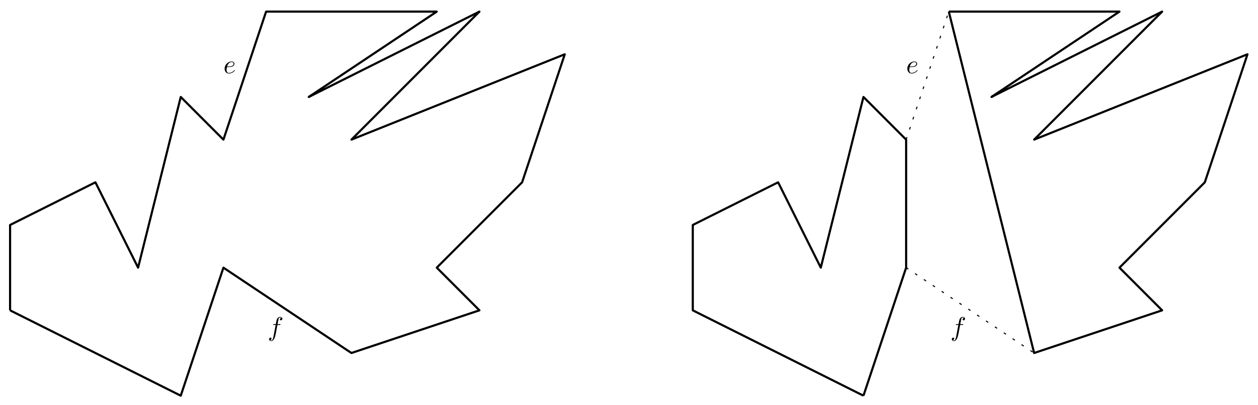

Figure 4.2: A hole polygon split into two hole polygons along a suitable pair of edges e and f.

4.1.1 A Hole Generated from a Polygon

Auer in his thesis [5, Lem. 2.11, p. 74] proved the following lemma for a set of points S.

Lemma 4.1.1. Auer's Lemma:
Points that lie on the boundary of the convex hull of S must appear in the polygonal chain in the same relative order as on the hull.

The following corollary is an extension of that lemma.

Corollary 4.1.2. Given a simple polygon P and its convex hull polygon C, any two adjacent vertices in C will be connected by two chains in P, one that contains all other vertices of C and another that contains no other vertices of C.

As the second chain contains no other vertices of C, the vertices of the chain between the endpoints lie completely inside the convex hull, so we call this type of chain an **inner** chain of P.

Definition 4.1.2. Inner chain. Given a polygon P and the convex hull polygon C of P. An inner chain of a polygon P is a polygonal chain in P that has two adjacent vertices of C as endpoints and no other vertices of C in the chain.

There are two degenerate cases of inner chains: i) a chain with four or fewer vertices can not be used to generate holes, and ii) a chain where all vertices between the endpoints are collinear can not be used to generate holes.

Forming a polygon Q from an inner chain of a simple polygon P gives us a polygon that encloses an area which is exterior to P. Q will also be in the closure of the convex hull polygon C of P. If the function Split a Polygon is used on Q along suitable edges of Q the resulting

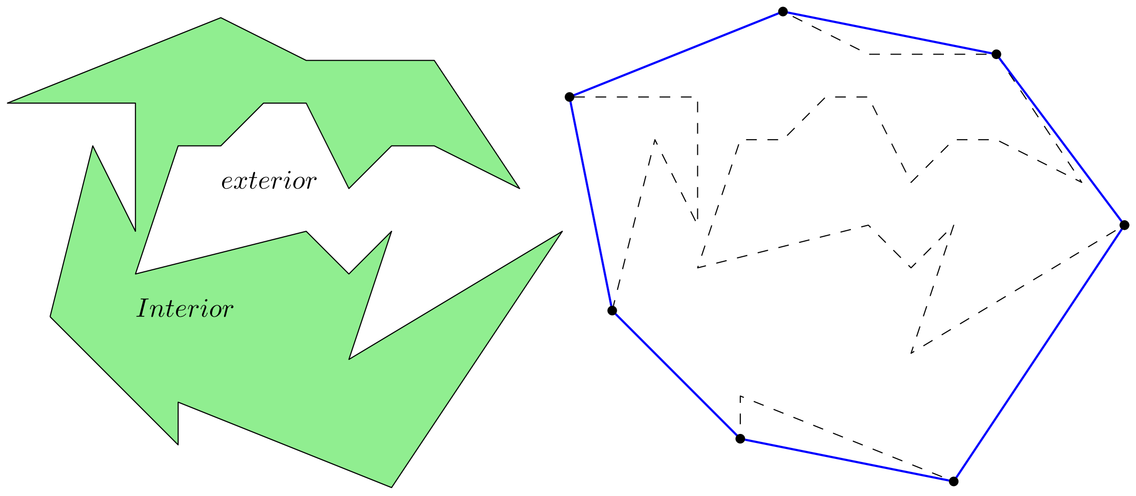

Figure 4.3: Polygon P (left) and its convex hull polygon C (right).

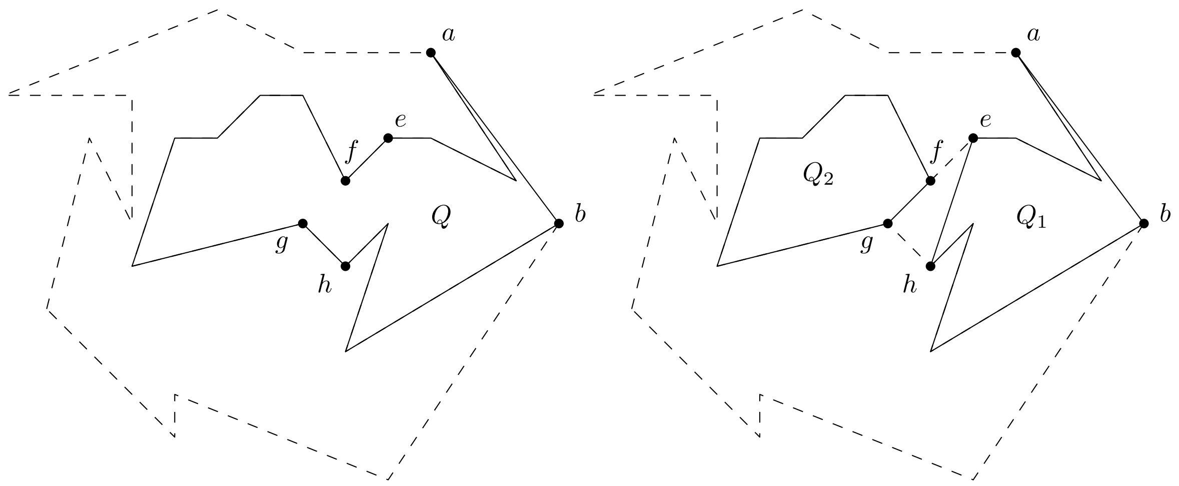

Figure 4.4: Polygon Q and resulting polygons Q_1 and Q_2.

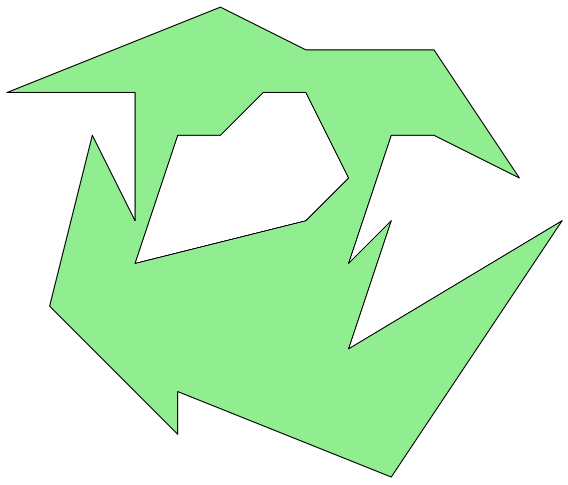
Figure 4.5: P split along the same suitable edges $[e, f]$ and $[g, h]$ results in a modified polygon P' and the new hole.

two simple polygons lie in the closure of C. If P is split along the same pair of edges the result is a nested polygon inside a modified polygon P', see Figure 4.3 to Figure 4.5.

Theorem 4.1.3. Given a polygon P, and a polygon Q formed from an inner chain of P which contains a suitable pair of edges.

Splitting P along the suitable pair of edges of Q, results in a polygon P' and a hole Q' in P'.

Proof. Given a polygon P, a convex hull polygon C of P, and a polygon Q formed from the vertices of an inner chain of P.

By the Jordan Curve theorem the interior of Q is outside of the interior of P.

Assume that a suitable pair of edges exist in Q, those edges then also exist in P.

By definition of the suitable pair and the function Splitting a Polygon, A suitable pair of edges of Q splits Q into two polygons.

As the edges of the suitable pair exist in P, splitting P along the same pair of edges results in splitting P into two chains, one chain does not contain a single vertex of C, the other chain contains all vertices of C. Both chains are inside the closure of C, so when they are closed, one forms the polygon P' and the other must then exist completely inside P' as a hole Q'. $\square$

The interior of P' is a combination of three components: i) the original interior of P, ii) the hole Q', and iii) the area of a quadrilateral bounded by the endpoints of the suitable pair of edges. The original interior of P is thus unmodified in the generation of a hole.

Similar can be said about splitting a hole. The hole Q already exists in the interior of polygon P. Modifying Q by splitting it by a suitable pair of edges forms two new holes Q_1 and Q_2, thus $P \setminus Q$ before and after the splitting of Q is unmodified.

The generation of a hole leaving the original interior of the polygon alone means generating holes can be done iteratively, i.e. the result of one generation can be fed into the next generation. A user can also modify the polygon by i) removing any chains that should not be used for generating holes, then adding them back in after the generation, or ii) if any nested holes should not be selected for modification, those holes can be removed before generating holes. If only nested polygons should be used to generate more holes, the user can use a convex hull polygon with the nested polygons to be modified in its interior.

Algorithm 6 shows the process of finding an inner chain or nested polygon to generate a new hole. No vertex is shared between chains so if n is the number of vertices in the polygon, and

Algorithm 6 Gather Inner Chains and Holes

```
1: Generate the convex hull polygon $C$ of polygon $P$.
2: For each pair of adjacent vertices in $C$ do
3:     Find the inner chain $I$ between the pair.
4:     If $I$ has 5 or more points and is not collinear then
5:         Add $I$ to set of usable chains.
6: For each hole $Q$ in polygon $P$ do
7:     If $Q$ has more than six vertices then
8:         Add $Q$ to set of usable holes.
9: Randomly pick one vertex $r$ from either the set of usable chains or the set of usable holes,
   where each vertex has the same likelihood.
10: The chain or hole that contains $r$ is then selected to generate a hole.
```

if the implementation to generate the convex hull polygon is $\mathcal{O}(n)$, then the whole process is at most an $\mathcal{O}(n)$ operation.

4.2 Using the Sweep Line Algorithm to Generate Holes

Our heuristic will use the Sweep Line algorithm as a basis to find adjacent edges in the SLS that will be considered as candidates for a suitable pair of edges. A vertical sweep-line will be swept over the x-axis and will generate holes from the suitable pairs of edges that are discovered.

The suitable pair of edges that are discovered in a single sweep will be a subset of all suitable pairs of edges, i.e. pairs with their locally interior sides adjacent in the SLS. A random rotation of the polygon before generating holes would allow other suitable pairs of edges to be detected.

Theorem 4.2.1. Given a polygon Q with suitable pairs of edges, a sweep line ℓ, and a random rotation α of Q and ℓ in relation to each other, all suitable pairs of edges of Q have a non-zero probability to be detected in the SLS.

See Figure 4.6 for an example for the following proof.

Proof. Given a polygon P and a suitable pair of edges e and f with endpoints e_a, e_b and f_a, f_b.

By definition of a suitable pair of edges e and f, there must be at least three interior diagonals between e and f that make up two triangles, each with two interior diagonals as sides.

W.l.o.g. assume the sweep line ℓ is rotated by a random angle $\alpha \in [0, 2\pi)$ before the sweep starts, then it is swept over P. No matter the value of α, all points are swept over, so we can consider the rotational value of α with ℓ fixed in an arbitrary point of the polygon.

Case 1: W.l.o.g. endpoint f_b is on the interior side of e, and at least a segment of f is visible on the interior side of e from f_i to f_b. The segment is defined by the angle ${}_{e_b}\angle_{f_i}^{f_b}$.

As f_b is on the interior side and e, f is a suitable pair, the angle ${}_{f_b}\angle_{e_a}^{e_b}$ is a non-zero angle. Fixing ℓ in f_b, ${}_{f_b}\angle_{e_a}^{e_b}$ represents a sub-interval which α can be drawn from. When α is drawn from this interval, ℓ crosses both e and f at the same time when sweeping over P.

Fixing ℓ in e_b, the angle ${}_{e_b}\angle_{f_i}^{f_b}$ represents another sub-interval that α can be drawn from.

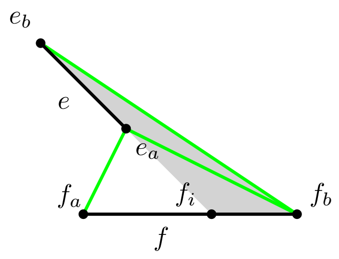

Figure 4.6: Pair of edges e and f in black, their interior diagonals in green, and the visibility of a segment of f from endpoint e_b in gray.

These angle intervals cannot be zero because the angles in the triangles with interior diagonals as sides cannot be zero, so any suitable pair must be discoverable with a rotation that matches at least one of the intervals for the pair.

Case 2: When the quadrilateral made up of e and f is convex, the angles ${}_{e_a}\angle^{f_a}_{f_b}$, ${}_{e_b}\angle^{f_a}_{f_b}$, ${}_{f_a}\angle^{e_a}_{e_b}$, or ${}_{f_b}\angle^{e_a}_{e_b}$ all independently define a non-zero subinterval that α can be drawn from. □

Even though hypothetically all suitable pairs are detectable, it is not known whether there are suitable pairs to be detected in a given polygon, or how many pairs will be detectable in a single sweep. The decision is left up to the user to choose i) whether to rotate the given input polygons, ii) how many holes to generate per rotation, and iii) what the maximum number of rotations should be. See Section 4.3 for the rotation implementation.

Definition 4.2.1. A **candidate** for a suitable pair of edges. Given adjacent edges a and e in the SLS with their interior sides facing each other, the pair form a candidate of a suitable pair. A candidate will be represented as (a, e).

Lemma 4.2.2. Given adjacent edges a and e in the SLS with interior sides facing each other, where a is inserted into the SLS in an earlier event point than e, candidate (a, e) is found when processing e.

Proof. Given a simple polygon P, and the insertion of an edge e of P into the SLS at event point j, assume e finds two edges a and b of P adjacent in the SLS.

P can be oriented such that the left side of the edges going from v_i to v_{i+1} is the interior side.

If v_i of e is the e_1 endpoint we know that edges above e are on the interior side, and if v_i is the e_2 endpoint we know that the edges below e are on the interior side.

The SLS uses the $\prec$ comparator, so we know the order of the edges in the SLS, so we know if a or b is on the interior side.

W.l.o.g. assume a is on the interior side. Checking if e is on a's interior side in the same way we checked if a is on the interior side of e, verifies that both edges have facing interior sides, and thus discovers the candidate (a, e) when processing e. □

Theorem 4.2.3. Given a line sweeping over a polygon with n number of edges in total, the maximum number of candidates is n.

Proof. When processing an event point up to two edges can be inserted into the SLS. After insertion, when processing edge e, e finds one adjacent edge a on its interior side in the SLS.

The candidate (a, e) will be counted at the processing of e, which means for n edges in total there will be n counts, or at most n candidates for all n edges. □

A degenerate case of a candidate is when an event point introduces two edges into the SLS at the same time that face each other's interior sides. By definition they share an endpoint and so cannot be used to cut the polygon. A minimum of one point in the chains on either side of the candidate are needed to make a hole.

Candidates have to be checked if there are any vertices or edges of Q that intersect the diagonals.

There are two categories of event points that could introduce points or edges that intersect a diagonal of a candidate:

1. After the insertion of the earlier edge a of a candidate (a, e), event points processed can introduce and remove edges or vertices of Q that intersect a diagonal of (a, e) before e is inserted into the SLS.
2. While a and e are adjacent in the SLS, event points can insert edges that are inserted in between a and e in the SLS.

Lemma 4.2.4. If an event point i exists inside the triangle with sides made up of an edge e and the diagonals to an event point j, there must exist edges that either intersect the diagonals, or the diagonals are not interior diagonals.

Proof. Given an edge e, and two event points i and j processed while e is in the SLS.

W.l.o.g. assume event point i is inside the triangle made up of edge e and the diagonals to event point j.

As all vertices of the polygon must be connected, an event point i inside the triangle between edge e and event point j must then be connected to both the endpoints of e as well as to j through a polygonal chain. Assume both diagonals are non-intersecting interior diagonals.
Case 1: The chain from an endpoint of e to i to j should be exterior to the triangle, however as i is interior to the triangle, this is a contradiction. This means one of the diagonals cannot be an interior diagonal if an event point is inside the triangle.
Case 2: If the whole chain from one endpoint of e to i then to j is inside the triangle, then one of the diagonals must be exterior to the polygon; a contradiction. □

Lemma 4.2.5. Given an edge a, a line sweeping from one endpoint of a to the other, and an ordered set of event points L that are i) inside the bounded area of a and ii) on the interior side of a.

The smallest angle $\angle_a$ between a and the first $j-1$ processed events of L out of n total events, $j \leq n$, can tell us if a processed point exists inside the triangle formed by a and the next event point l_j to be processed.

Proof. The event point set L contains n points: $L = \{l_0, l_1, ..., l_{n-2}, l_{n-1}\}$.

Case 1: assume an ascending direction of the sweep-line, which starts the sweep line in a_1

The first processed event point l_0 after a_1 defines a triangle with a. The angle between $[a_1, a_2]$ and $[a_1, l_0]$ defines an angle ${}_{a_1}\angle_{a_2}^{l_0}$.

Processing event points $\{l_0, l_1, ..., l_{j-1}\}$ where $j \leq n$ updates the smallest angle $\angle_a$ seen between a and an event point in $\{l_0, l_1, ..., l_{j-1}\}$. Assume an event point l_i where $i < j$ defines the smallest angle $\angle_a$.

If $\angle_a$ is smaller than the angle between ${}_{a_1}\angle_{a_2}^{l_j}$, as all event points $\{l_0, ..., l_i, ..., l_{j-1}\}$ are to the left of l_j, then l_i is inside the triangle of endpoints of a and diagonals to event point l_j.

Case 2: assume a descending direction of the sweep line, which starts the sweep line in a_2.

The first processed event point l_{n-1} after a_2 defines a triangle with a. The angle between $[a_2, a_1]$ and $[a_2, l_{n-1}]$ defines an angle ${}_{a_2}\angle_{a_1}^{l_{n-1}}$.

Processing event points $\{l_{n-1}, l_{n-2}, ..., l_{j+1}\}$, $j \geq 0$, updates the smallest angle $\angle_a$ seen between a and an event point in $\{l_{n-1}, l_{n-2}, ..., l_{j+1}\}$. Assume an event point l_k where $k > j$ defines $\angle_a$.

If $\angle_a$ is smaller than the angle between ${}_{a_2}\angle_{a_1}^{l_j}$, as all event points $\{l_{n-1}, l_{n-2}, ..., l_{j+1}\}$ are to the right of l_j, then l_k is inside the triangle of endpoints of a and diagonals to event point l_j. $\square$

Figure 4.7 is a visual example for the proof. A candidate (a, e) found at the insertion of e in event point j can be immediately invalidated before it is considered in one direction. Candidates are gathered from sweeping in both directions, and if there are two identical candidates counted, as a candidate can only be inserted once in a single sweep, it must have been counted once in each sweep.

See Appendix A.1 for the function that calculates the angle between the adjacent edge and the event point.

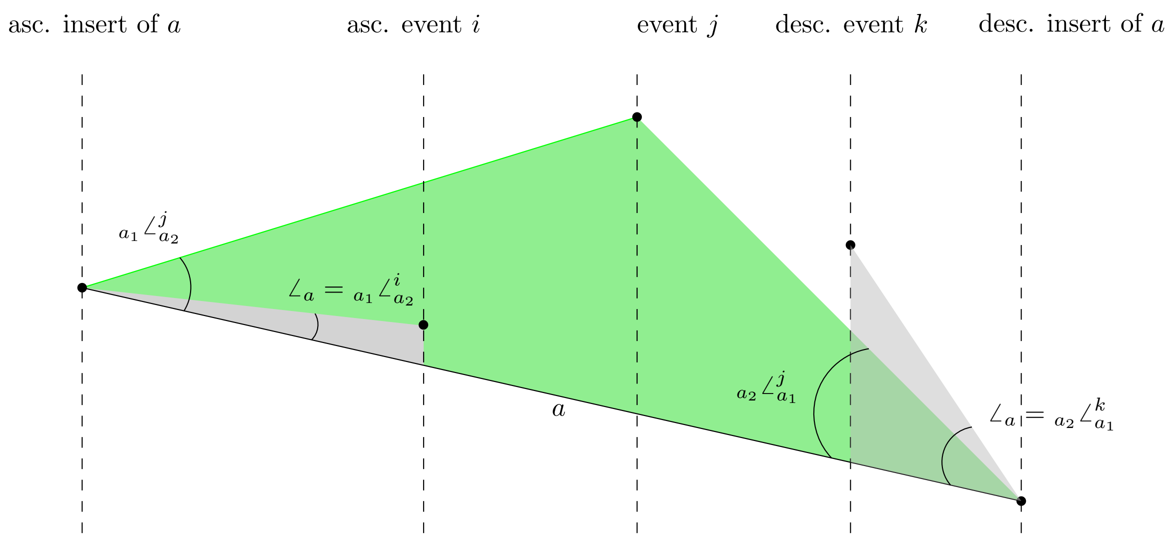
asc. insert of a
asc. event i
event j
desc. event k
desc. insert of a
${}_{a_1}\angle^{j}_{a_2}$
$\angle_a = {}_{a_1}\angle^{i}_{a_2}$
${}_{a_2}\angle^{j}_{a_1}$
$\angle_a = {}_{a_2}\angle^{k}_{a_1}$
a

Figure 4.7: The triangle created by the adjacent edge a in the SLS and event point j (green), and the smallest angle $\angle_a$ (gray) found in an event point i in the ascending sweep and found in an event point k in the descending sweep.

Lemma 4.2.6. Given two adjacent edges a and b in the SLS that form a candidate (a, b), any event point that inserts edges between a and b invalidates (a, b) as a suitable pair of edges and the candidate can be removed from found candidates.

Lemma 4.2.5 invalidates candidates at the insertion of the edge e of a candidate (a, e), and Lemma 4.2.6 invalidates candidate (a, b) at the insertion of an edge e between the edges of a and b. Candidates that survive passing the line-sweep from both directions must be suitable pairs of edges.

Theorem 4.2.7. Sweeping a line ℓ over a simple polygon P, assuming that suitable pairs of edges exist in P, we can find all suitable pairs of edges that cross ℓ at the same time, and use a discovered suitable pair to modify P to generate a hole.

This follows from Theorem 4.1.3, Lemma 4.2.5 and Lemma 4.2.6.

Finding the suitable candidates that cross the sweep line at the same time takes two sweeps, so time complexity of the sweeps is $\mathcal{O}(n \log n)$ for n vertices. In a sweep there are maximum n candidates considered, so memory complexity of a sweep is $\mathcal{O}(n)$.

Algorithm 7 describes the process for finding candidates and turning them into suitable pairs.

Algorithm 7 Sweep Line Process for Finding a Suitable Pair

```
1:  Initialise a set $T$ of candidates and a set $U$ of suitable pairs.
2:  For each point $j$ in event queue; ascending do
3:      Insert, or find the two edges in the SLS that are adjacent to event point $j$.
4:      If edge $e$ was inserted into the SLS then
5:          Initialise a set of candidates associated with $e$.
6:          If adjacent edges $b$ and $c$ of $e$ in the SLS have a candidate $(b, c)$ then
7:              Remove candidate $(b, c)$.
8:          Get the adjacent edge $a$ on the interior side of $e$ in the SLS.
9:          If $\angle_a > {}_{a_1}\angle^{j}_{a_2}$ then
10:             Add candidate to a set associated with $a$.
11:             Update $\angle_a$ to ${}_{a_1}\angle^{j}_{a_2}$.
12:     If edge $e$ should be removed from the SLS then
13:         If the set of candidates associated with $e$ is not empty then
14:             Add candidates to the set $T$ of candidates.
15:         Get the adjacent edge $a$ on the interior side of $e$ in the SLS.
16:         If $\angle_a > {}_{a_1}\angle^{j}_{a_2}$ then
17:             Update $\angle_a$ to ${}_{a_1}\angle^{j}_{a_2}$.
18: For each point $j$ in event queue; descending do
19:     Insert, or find the two edges in the SLS that are adjacent to event point $j$.
20:     If edge $e$ was inserted into the SLS then
21:         If adjacent edges $b$ and $c$ of $e$ in the SLS have a candidate $(b, c)$ then
22:             Remove candidate $(b, c)$.
23:         Get the adjacent edge $a$ on the interior side of $e$ in the SLS.
24:         If $\angle_a > {}_{a_2}\angle^{j}_{a_1}$ then
25:             Add candidate to a set associated with $a$.
26:             Update $\angle_a$ to ${}_{a_2}\angle^{j}_{a_1}$.
27:     If edge $e$ should be removed from the SLS then
28:         If the set of candidates associated with $e$ is not empty then
29:             Add candidates to the set $T$ of candidates.
30:         Get the adjacent edge $a$ on the interior side of $e$ in the SLS.
31:         If $\angle_a > {}_{a_2}\angle^{j}_{a_1}$ then
32:             Update $\angle_a$ to ${}_{a_2}\angle^{j}_{a_1}$.
33: Sort the candidates in $T$.
34: If a candidate $(a, b)$ has two instances in $T$ then
35:     Add $(a, b)$ as a suitable pair to $U$.
36: Pick a random suitable pair from $U$ to generate a hole.
```

4.3 Implementing a Rotation of a Point Set

A rotation of a point set using floating point values gives a high likelihood of a perturbation of the coordinate values, and for some simple polygons that would mean they wouldn't be simple anymore. This is then a problematic challenge even if a user of the code would try and use arbitrary precision to rotate a point set exactly, as the code accepts at most double floating point precision values.

A perturbation of the points affects i) the order in the event queue, ii) the order of the line segments in the SLS, and iii) any angle calculations done. To combat this we assume that the point set coordinates given are the true coordinates that have not been rotated and implement a version that rotates the point set and sweeps the rotated polygon.

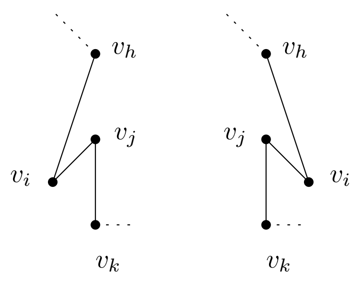

Figure 4.8: Vertex v_i has switched halfplanes of edge $[v_j, v_k]$.

After rotating the given point set, as a pre-processing step, we check if the polygons with the rotated point set is still simple. Only checking if the polygon is simple would not catch the case where a vertex has switched half-planes compared to its adjacent edges in the polygon, see example in Figure 4.8. This means we also check for an edge e in the polygon if the adjacent vertex to e has switched half-planes.

The event queue is ordered lexicographically given the rotated point set of the polygon. The event points could be processed out of the correct order in relation to the rotation, to try and combat this, at each event point we check i) if the first edge in the SLS has its interior side facing the rest of the polygon, and ii) if the interior/exterior sides of the inserted edges are correct compared to the adjacent edges in the SLS.

The perturbation can also affect angle calculations. To combat this we calculate any angles by using the original point set rather than the perturbed point set.

Algorithm 8 describes the additions to the Sweep Line process described in Algorithm 7 for the rotated implementation. This implementation enables the rotation of many point sets for the generation of holes, but a subset of all point sets will have problems with the rotation. Then it is still possible to use the no-rotation version over the original point set, or over the original point set with the x and y coordinates swapped.

Algorithm 8 Added Rotational processes

```
1: Copy the point set $S$ to a new set $R$ and rotate the points of $R$ by a random angle in the
   interval $[0, 2\pi]$.
2: If polygon $P$ is not simple in $R$ then
3:     Stop the process and return a rotation error.
4: For each edge $e$ in $P$ do
5:     Get the edge $e$ and adjacent vertex $v$ from $Q_R$ and $Q_S$.
6:     If $v$ is not in the same half-plane in both $Q_R$ and $Q_S$ then
7:         Stop the process and return a rotation error.
8: Get the convex hull $C$ of polygon $P$ in $S$.
9: Get the inner chains of $P$.
10: Randomly select an inner chain or a hole to work with.
11: Form a polygon $Q_R$ from $R$ and $Q_S$ from $S$.
12: Form an event queue for $Q_R$.
13: For each event point $j$ in the event queue (ascending/descending) do
14:     If the first edge in the SLS has its interior side facing the wrong way then
15:         Stop the process and return a rotation error.
16:     Insert or find the edges incident to $j$.
17:     If an incident edge of $j$ cannot be found in the SLS then
18:         Stop processing and return a rotation error.
19:     Get the adjacent edges $a$ and $b$ of incident edge $e$ in the SLS.
20:     On top of normal processing:
21:     If the sides of an edge $e$ are facing the wrong way compared to $a$ or $b$ then
22:         Stop processing and return a rotation error.
23:     If an insertion or a removal needs to calculate an angle then
24:         use the $Q_S$ polygon to calculate any angles necessary.
```

CHAPTER 5

ANALYSIS OF THE ALGORITHMS

The datasets that will be used to analyse the algorithms will be i) real-world datasets, and ii) a dataset of points with integer coordinates on a grid. The real-world datasets were graciously provided by my supervisor Martin Held and are point sets of circuit boards, outlines of objects, outlines of text symbols and various other real-world data sets that have not been specifically prepared for the tests. The smallest real-world dataset contained 1024 points, and the largest $629\,248$ points, and each dataset has a distinct number of points.

There are a few problems with using real-world data:

1. The data sets are not open and so cannot be published.
2. The properties of the data sets are uncontrolled and wildly different which can show strange, and unexplained outliers in the graphs.

However the benefits of using real-world data sets are:

1. They are a good indicator what the results of using the algorithms on real-world data sets would look like.
2. They verify that the code works on complicated real-world data sets.

The real-world dataset will be used for timing of the variations as well as for analysis of the hole algorithm.

The grid-point datasets are sets of $n \times n$ points with integer coordinates distributed uniformly in a square. It will be used in analysing any distributions in the random generation of simple polygons in the variations.

Python scripts were used to run the program and to gather the data into an SQLite database. The machine used for the timing measurements was an Intel Core i7-8700 CPU with 3.20 GHz and 32 GB of memory.

5.1 Simple Polygon Generation

This section will answer two main questions: i) do the variations increase the time efficiency of the 2-Opt line-sweep heuristic, and ii) is there a difference between the variations in regard to the random generation of polygons.

5.1.1 Timings of the variations

The real-world datasets were in total 436 and had from 1024 to $629\,248$ points. Five simple polygons were generated for each dataset with the same five seeds chosen at random in the range $[1, 10^{17} - 1]$ to initialise the random generator.

Comparison to RPG

RPG 2-Opt generator from Thomas Auer [5] was used as a comparison as it is the gold-standard in generating simple polygons. RPG 2-Opt was timed with Python, i.e. outside of the RPG code itself, but the variations were all timed with C++ from inside the code. The variations were timed from when the sweep line started to when it ended, leaving out handling the input or the output of the code. This means the comparison is slightly off in favour of the variations.

Algorithm	Min	Max	Average	Weighted average
Reset to zero	0.048	0.119	0.065	0.059
Reverse	2.151	24.178	6.813	6.131
Repeat index	3.138	9.231	4.988	4.489
Continue	2.039	6.995	3.508	3.157
Random	0.512	5.031	1.312	1.181

Table 5.1: The speed increase of the variations compared to RPG 2-Opt.

Table 5.1 compares the ratio of the time taken between the variations and RPG 2-Opt. In an attempt to gauge the difference between timing in C++ and timing in Python, timing tests were done on the reverse algorithm. For a file with 1088 points the difference was 3 milliseconds, or 12% of the process time. For a file with $100\,240$ points, the difference was 2 seconds, or 7% of the process time. To account for this estimated 10% difference between RPG 2-Opt and the variations, the column "Weighted average" in table 5.1 is the result of the "Average" column with a 10% reduction of the RPG 2-Opt timings.

The timing results are in Figure 5.1. Reset to zero is the slowest variation, and the Reverse is the fastest variation for point sets larger than 8000 points. Three of the variations are consistently

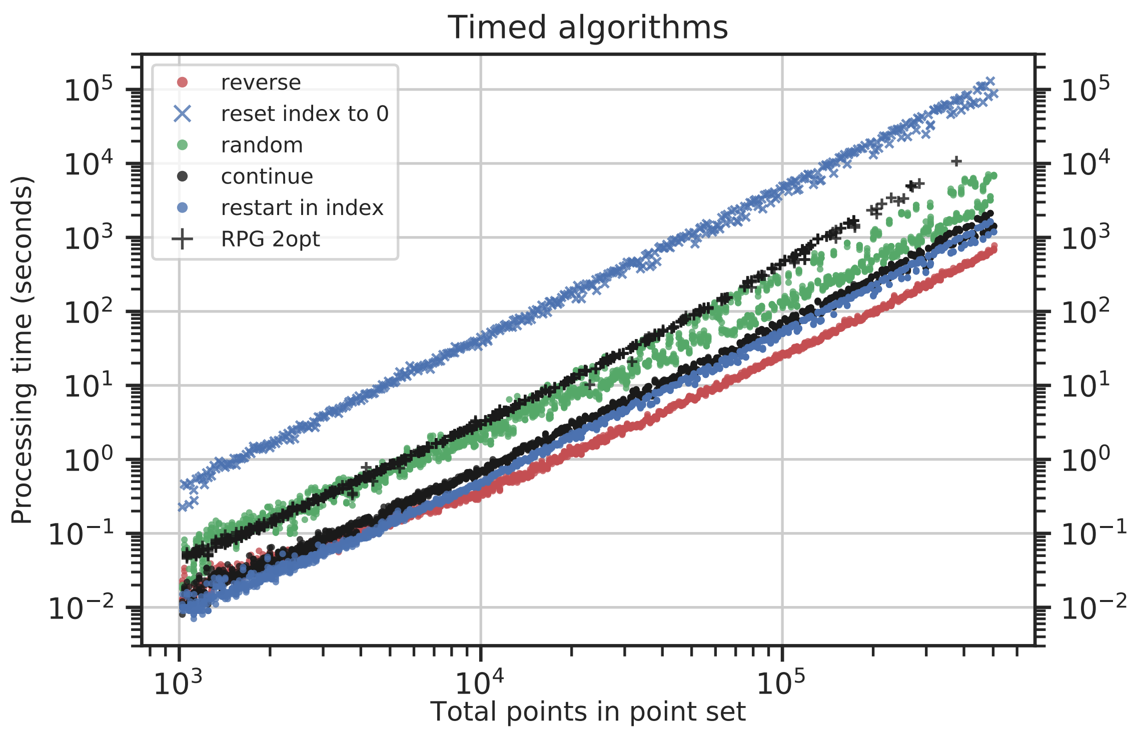

Figure 5.1: Timing graph.

quicker than RPG 2-Opt and even Random is quicker than RPG 2-Opt after a certain point set size.

From the figure and the table we see that three of the variations do offer a distinct speed increase of the heuristic.

Following up on the discussion we had in regard to saving the SLS for the Reverse variation (see Section 3.4.2), Figure 5.1 shows that, except for the Random variation and RPG 2-Opt, the slope of the lines are very similar and show (roughly estimated) a $\mathcal{O}(n^2)$ time increase based on n points in a point set. The deduction is that the Reverse variation can see $\mathcal{O}(m^2)$ new intersections for m untanglings of intersections. This verifies that the number of times we save the SLS to then not use it overshadows the duration of saving and reloading the SLS when it doesn't find an intersection.

Regarding RPG 2-Opt, in Figure 5.1 there are missing data points for RPG 2-Opt as it was not able to finish the generation for all data sets, most likely due to problems with collinearities of the real-world sets. This was a big problem, especially with larger data sets. Attempts were made to generate a simple polygon with the same seeds as the other variations, and if a seed returned an error, modify the seed up to 18 times to try and get a simple polygon with the modified version of the seed. If all of those failed, we deemed it a failure and moved on to the next file. This was only necessary for RPG, all of the variations finished successfully with the given first seed. There is also a slight upwards curve of RPG 2-Opt, which is possibly because of the extra processing of adding/removing event points for intersections in the Bentley-Ottman version of the sweep line status set.

5.2 The Distribution of the Randomly Generated Simple Polygons

Point sets of $n \times n$ points, with integer coordinates that are uniformly distributed in a square, show both line symmetry as well as rotational symmetry.

Running the variations to generate simple polygons on such a point set that is highly symmetric gives an opportunity to compare the distributions of the generated edges connected to a point between two symmetrically congruent points.

In this section we will develop a criterion to measure the randomness of the generated simple polygons of the variations in relation to each other.

The Divergence Metric

There are two distributions we are interested in:

1. For a point set where we can generate all known simple polygons, what is the distribution of the simple polygons, and how far away from a uniform distribution is it.
2. For a point set where the number of generated simple polygons are less than the set of total simple polygons, how far away from a uniform distribution are the generated simple polygons.

For the first case, we can compare the distribution of the generated n distinct simple polygons of the variations to a theoretical uniform distribution with probability $\frac{1}{n}$ for n total simple polygons.

For the second case, we can compare the distributions of generated line segments that are connected to symmetrically congruent points. Given a specific, but arbitrary point a and all line segments with a as an endpoint, and a symmetrically congruent point b and all line segments with b as an endpoint, we can count the number of simple polygons each line segment to a or b exists in out of all the generated polygons. If the distribution of simple polygons is uniform, then for each edge x in the set of edges $\mathcal{X}$ incident to a point, the probability $P(x)$ of point a

and $Q(x)$ to point b of the edges of the generated polygons should be approximately equal.

$$\sum_{x \in \mathcal{X}} |P(x) - Q(x)| \tag{5.1}$$

Equation 5.1 is the divergence metric we will use. If the distribution approximates a uniform distribution, the divergence metric should be minimal.

Generating Polygons on 3×3 *Grid Sets*

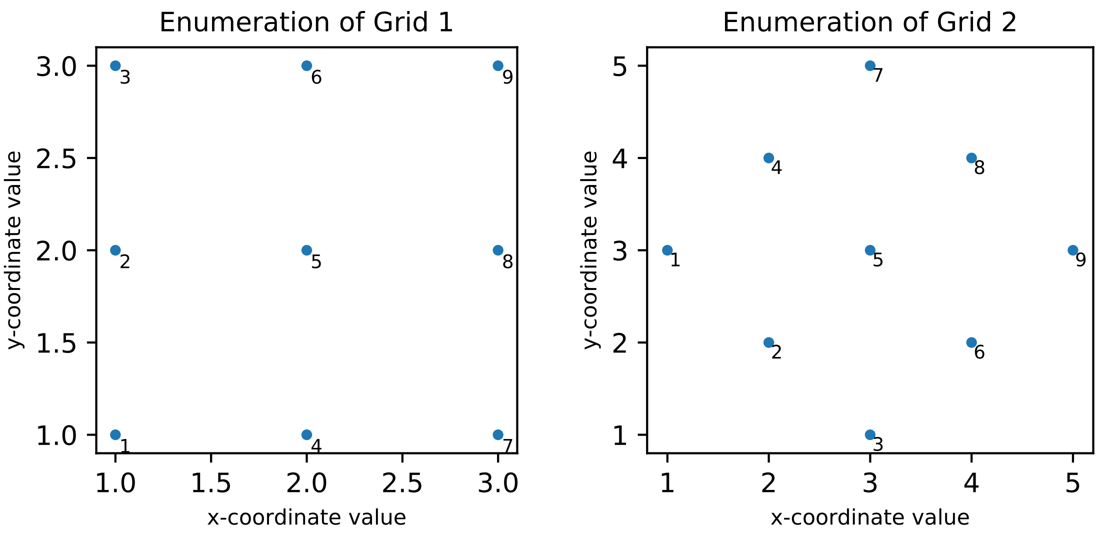

Figure 5.2: 3×3 grid sets.

Figure 5.2 shows the two grids that were used for the 3×3 sets. The lexicographical ordering of the two grids is different. Comparing the result of these two grids can tell us if there is a difference in the distribution of the random generation that is dependent on the rotation of the grid set, or in other words, dependent on the ordering of the points in the event queue.

The setup of the grids is such that only eight simple polygons exist for each grid, see Figure 5.3. The symmetry of the point set means that the eight polygons are all rotated or mirrored versions of each other.

With all simple polygons known, we can discover any difference in the random generation by generating $100\,000$ polygons and comparing the distributions of the generated polygons of the variations.

Figure 5.4 shows graphs of the generated simple polygons per variation and for both grid sets, where the left column of sub-figures are for grid 1, and the right column of sub-figures are for

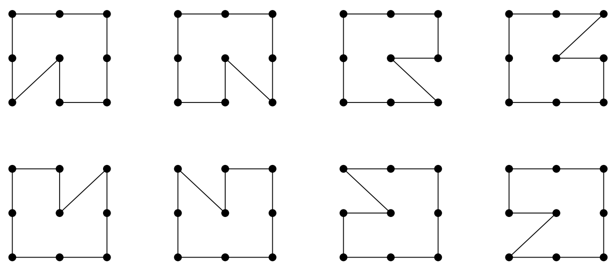
Figure 5.3: The eight simple polygons of the 3×3 grid displayed on Grid 1.

grid 2. The x-axis are the eight simple polygons enumerated, and the y-axis are the percentages out of total generated simple polygons each polygon was generated. Note that the enumerations of the polygons are only the same for the same grid.

There was a serious problem trying to generate simple polygons with the RPG 2-Opt code as it did not handle the collinearities in grid set 1 well. Out of $100\,000$ attempts only $54\,937$ simple polygons were generated for grid 1. This is with the up to 18 attempts to modify each attempted seed if the original attempt failed. We were able to get $100\,000$ generated polygons for grid 2 with the additional attempts.

Comparing the distributions of generated polygons for each variation to a perfectly uniform distribution (eight bins, each with $\frac{1}{8}$ probability) using the divergence metric, gives us the results in Figure 5.5. Out of the variations Random has the smallest divergence from a perfectly uniform distribution, and Reset to 0 had the highest divergence. RPG 2-Opt has a very high divergence for grid 1 and a very low divergence for grid 2, which shows that it is very sensitive to specific setups of the input point set.

Comparing Distributions of Edges to congruent Points

Figure 5.6 are two examples for the reverse variation of all the edges that exist in some generated simple polygon that are connected to a specific endpoint, the endpoint (1,1) in the left subfigure and (2,2) in the right subfigure. The percentages next to each edge represents the percentage of the generated polygons an edge exists in out of all the generated polygons for Grid 1.

As an example of applying the divergence metric calculation, let us compare the distribution of point (1,1) in Figure 5.6 to the line-symmetric distribution of point (1,1). Point (1,1) has the distribution $\{0.42946, 0.09958, 0.47096\}$, the line symmetric version is $\{0.47096, 0.09958, 0.42946\}$, applying the divergence metric: $|0.42946 - 0.47096| + |0.09958 - 0.09958| + |0.47096 - 0.42946| = 0.083$.

In our analysis the distribution of point (1,1) is compared with the distribution of points: (1,3),

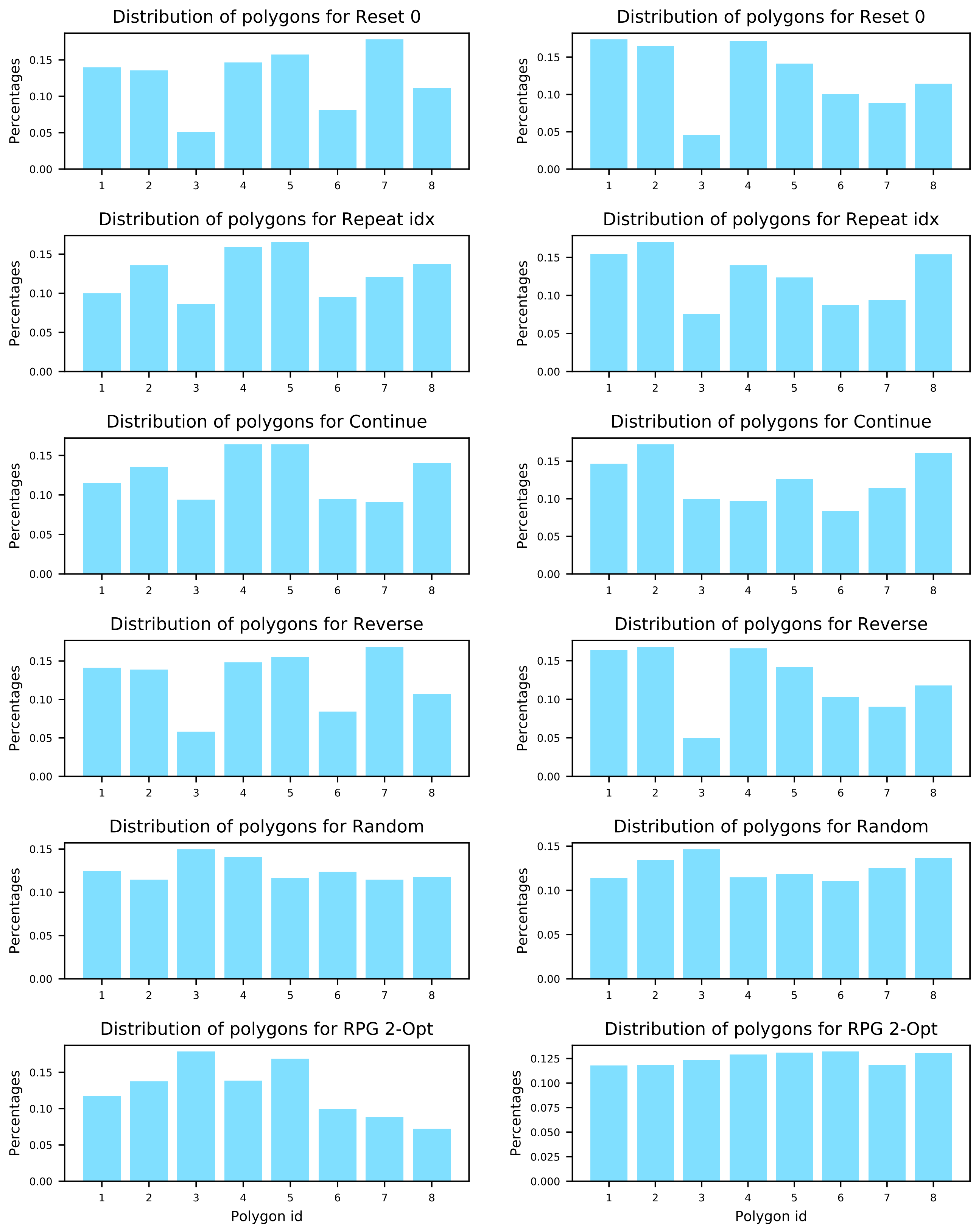

Figure 5.4: Generated simple polygons for the 3×3 grids.

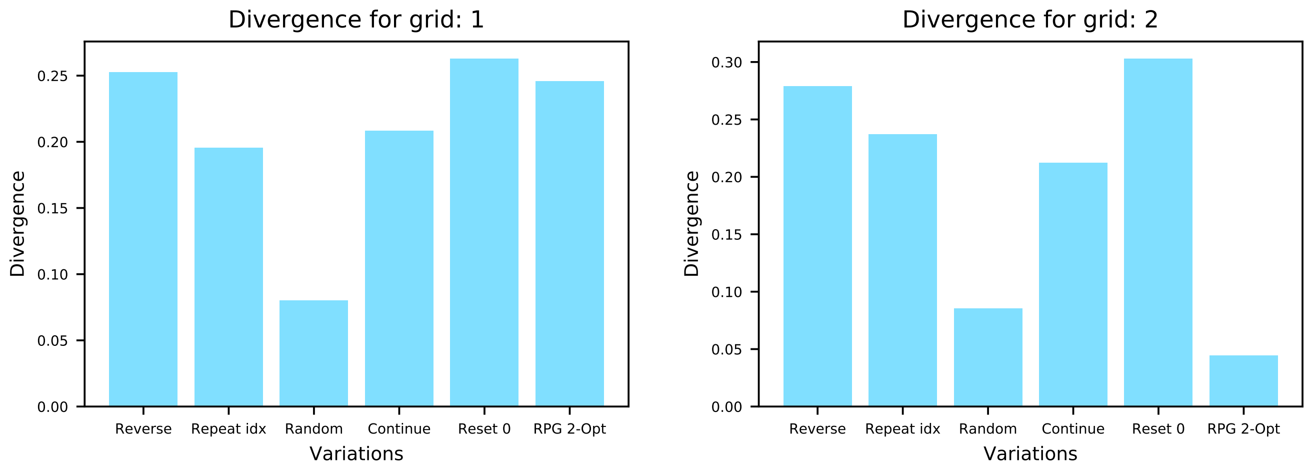

Figure 5.5: 3×3 divergence for the generated polygons.

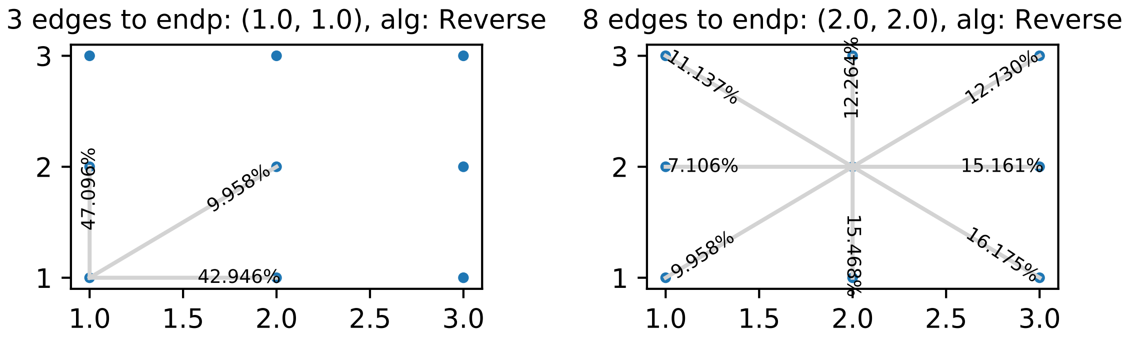

Figure 5.6: The distribution of edges connected to two points for the Reverse variation.

(3,1), and (3,3), as they have a congruent relationship with (1,1) to the rest of the point set.

For the right sub-figure of Figure 5.6 we use the original distribution in the divergence comparison with the line-symmetric (flipped) and rotated versions, i.e. the distribution at 0° gets compared to:
$\{0^\circ_{flipped}, 90^\circ, 90^\circ_{flipped}, 180^\circ, 180^\circ_{flipped}\}$.

Figure 5.7 shows the result of comparing the default distribution with the line symmetric or rotation symmetric distribution for the point in coordinate $(2, 2)$ in the right sub-figure of Figure 5.6. The slight offset from $0^\circ, 90^\circ, 180^\circ$ is because the sorting is done with the angles from $(-180^\circ, 180^\circ]$, and the offset makes sure that the edge that lines up with 180° is on the right side of the sort.

Figure 5.8 shows the average divergence for each group of congruent points. The figure also shows in "All" the zero divergence of the edge distributions of the generated eight distinct simple polygons.

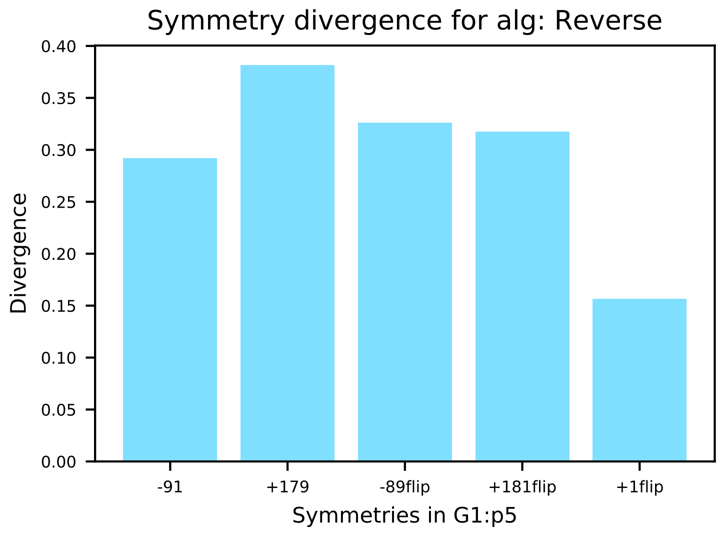

Figure 5.7: Divergence between the rotated/flipped distributions of point (2,2) and the original distribution at $0°$.

The relationship between the divergence of the generated polygons, see Figure 5.5 and the divergence of the symmetries of the edge distributions of the points in Figure 5.8 shows a clear correlation.

RPG 2-Opt shows a problem with collinear point sets that seems to be less if the collinearities are not in line with the coordinate axis.

5.2.1 Generating Polygons on 5×5 Grid Sets

Figure 5.9 shows the two grid sets used as 5×5 grid sets, the left sub-figure is Grid 1, the right is Grid 2.

500 000 polygons were generated for each variation using the same seeds for each grid. For the RPG 2-Opt we attempted to generate 100 000 polygons but only got 7495 for grid 1 and 38 954 simple polygons for grid 2. We did not generate more for RPG 2-Opt as it took much longer to generate just these specimens because of the errors than it took generating polygons for the variations.

Table 5.2 shows the number of unique generated simple polygons for the variations as well as the number of unique polygons for all variations.

Figure 5.10 shows the average divergence of each group of congruent points. In the two sub-figures we can see that "All" now shows a divergence value, which tells us that the generated simple polygons cannot be the complete set of simple polygons, and that the metric seems to

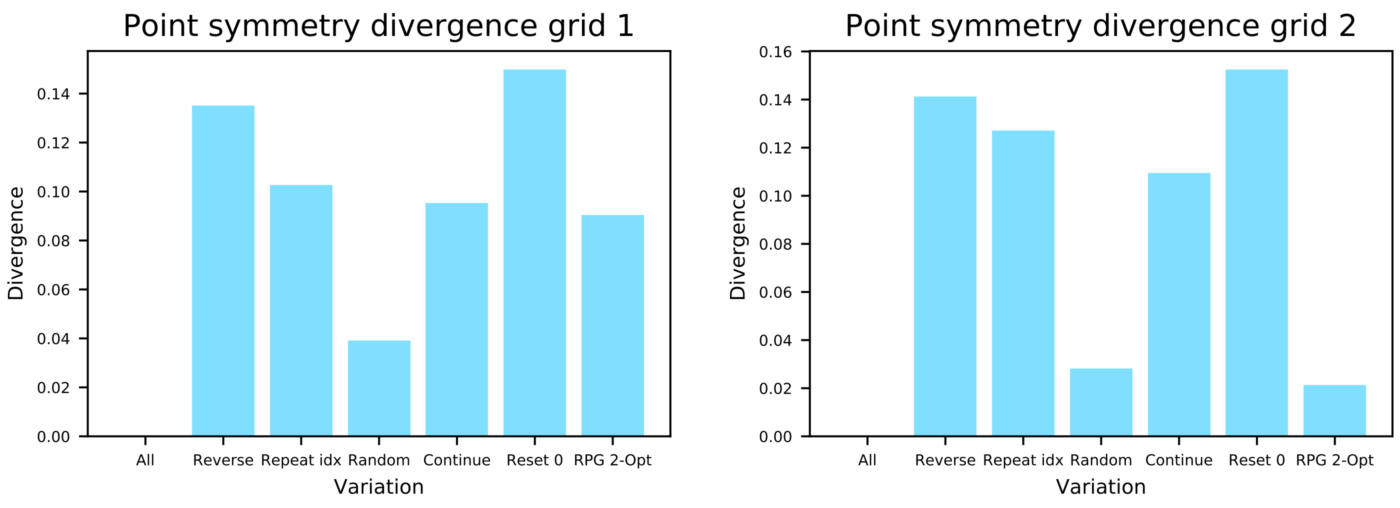

Figure 5.8: Symmetry divergence of the two 3×3 grids.

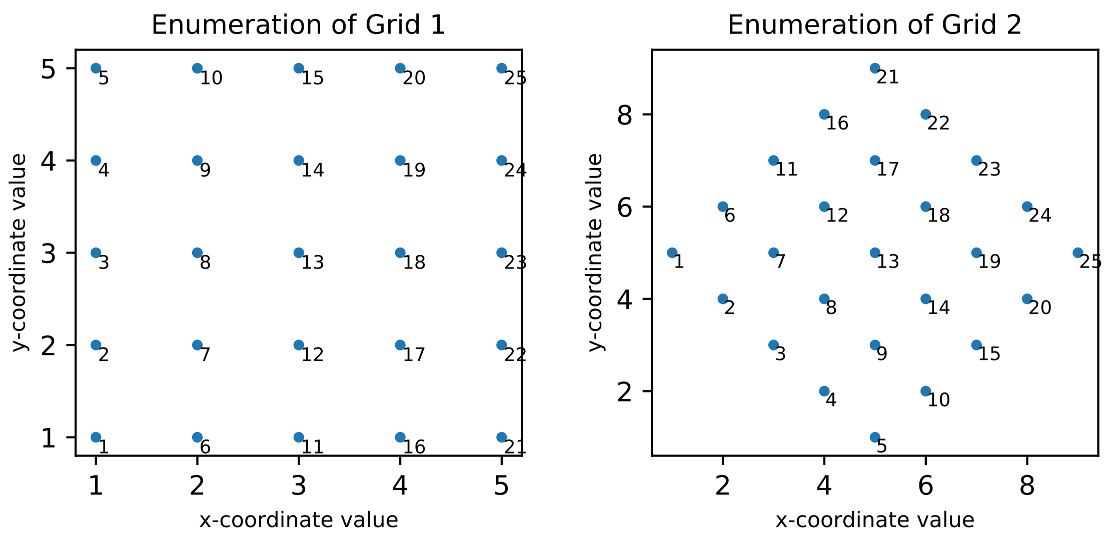

Figure 5.9: The two 5×5 grid sets.

catch the divergence in the distinct generated polygons.

The figure and the table do not correlate perfectly, as the table shows for grid 1 that continue has the most unique polygons, grid 2 shows Random having the most unique polygons, and the symmetry divergence seems to show that Continue has the lowest divergence for both grids.

We expect that a low divergence of the edge distribution means the edges on either side of a symmetric line of a point are equally likely, so it seems a much better metric than counting the uniqueness of the generated simple polygons in this specific example.

RPG 2-Opt shows a high divergence value for grid 1, and a low value for grid 2, which is

Algorithm	Grid 1: No. of unique polygons	Grid 2: No. of unique polygons
Reset to 0	212 047	277 407
Reverse	228 566	289 080
Repeat index	288 737	315 429
Continue	301 542	312 543
Random	288 044	319 711
All (var)	808 872	945 583
RPG 2-Opt	7238 / 7495	36 101 / 38 954

Table 5.2: Unique generated simple polygons for the 5×5 grids.

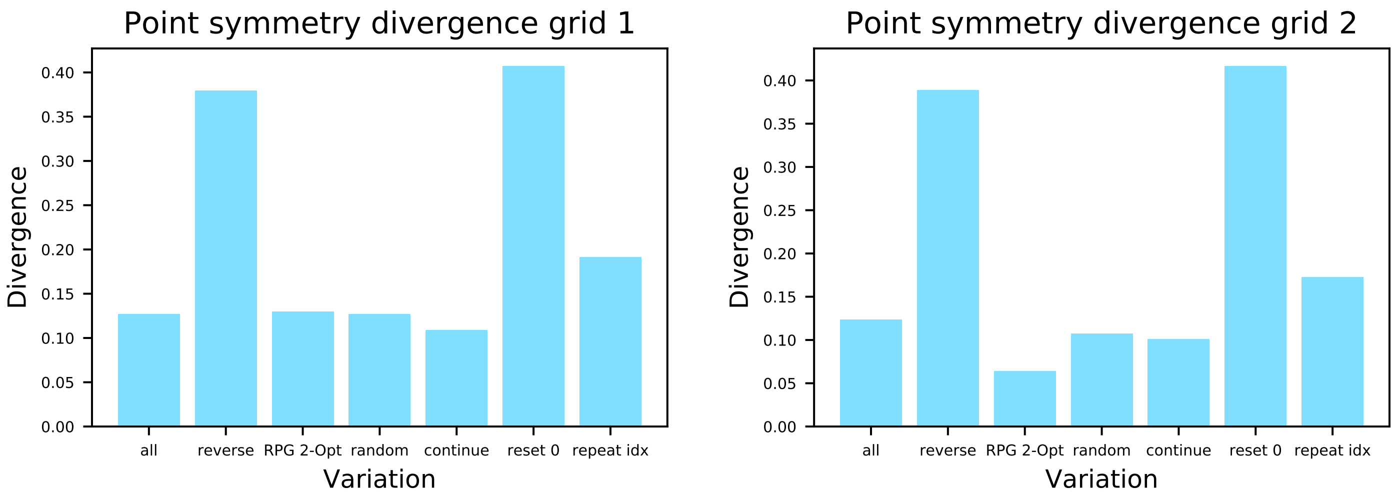

Figure 5.10: Symmetry divergence of the two 5×5 grids.

most likely connected to the number of generated polygons which is connected to its problems dealing with collinearities.

5.2.2 Generating Polygons on 7×7 Grid Sets

As a final comparison two 7×7 grid sets were analysed, see Figure 5.11. We do not use a comparison to RPG 2-Opt as the problems with it with the 5×5 set were too great already.

Algorithm	Grid 1: No. of unique polygons	Grid 2: No. of unique polygons
Reset to 0	499 282	499 968
Reverse	499 817	499 969
Repeat index	499 970	499 970
Continue	499 971	499 970
Random	499 969	499 970
All	2 496 003	2 495 645

Table 5.3: Unique generated simple polygons for the 7×7 grids.

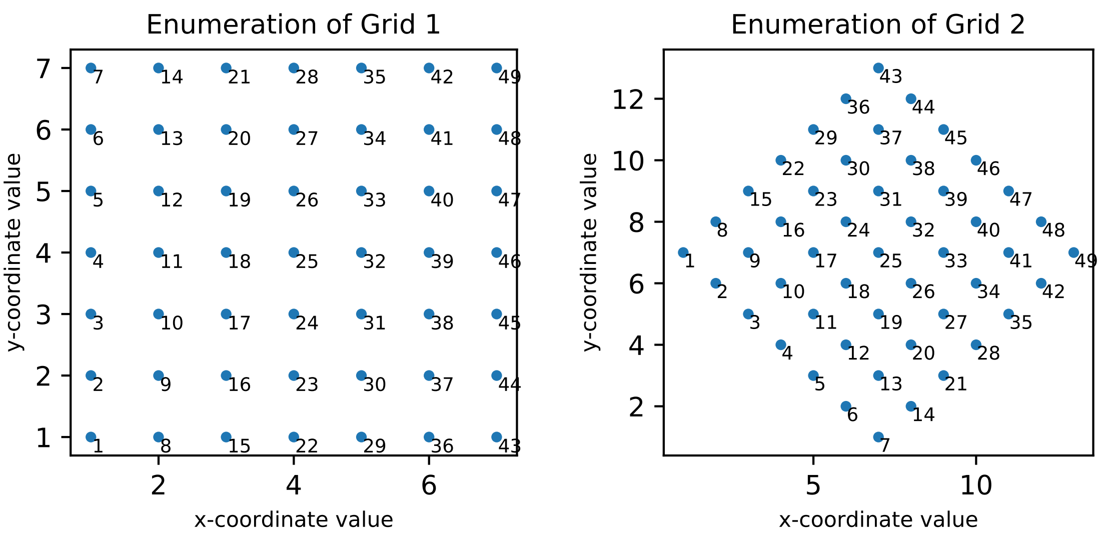

Figure 5.11: The two 7×7 grid sets.

For this test we generated $500\,000$ polygons for the different variations. Table 5.3 shows that once sets get over a certain (moderate) size, the number of unique generated simple polygons is not a good indicator of the randomness of the distributions.

Figure 5.12 shows that the divergence metric over the symmetries of the edge distributions is still useful as a metric for the difference between the distributions. The graphs show that Random and Continue have the lowest divergence, and the Reset to 0 and Reverse have the highest divergence, with Repeat index being in the middle.

Conclusion

We found that the Reverse variation was the fastest of the algorithms tested but also had the highest divergence value, so it is recommended only when speed is important.

The Random variation seemed consistently to have a low divergence value, but was overtaken by Continue for the 7×7 grid, hinting that once over a certain amount of points, the randomness of which intersection Continue finds becomes good enough to give a low divergence in the generation.

Random being the slowest of the four fastest variations means it won't get recommended except in the case of wanting the lowest divergence possible on input sets with very few points. With just a few input points there aren't enough edges in the SLS at the same time for Continue to not show a bias.

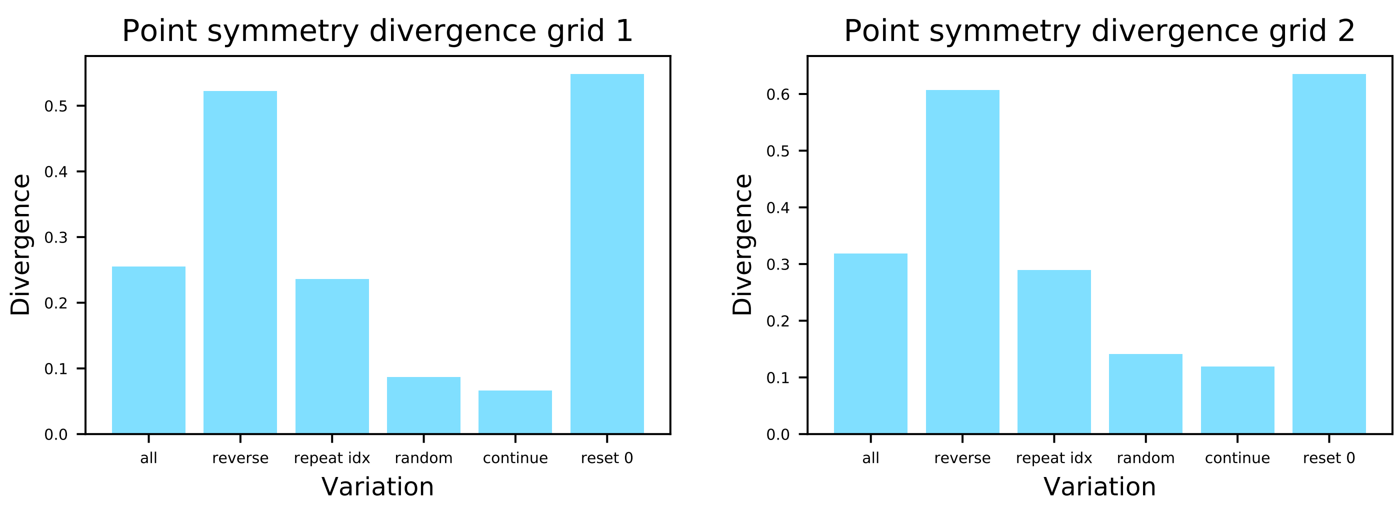

Figure 5.12: Point symmetry divergence of the two 7×7 grids.

Repeat in Index is in the middle regarding speed and divergence of the three recommended variations. It is 50% faster than continue on average, but has a higher divergence. It is 33% slower than Reverse, but it has lower divergence than Reverse.

RPG 2-Opt is still a noteworthy heuristic, and if the collinear problems can be fixed, or the input sets are guaranteed to have no collinearities, the small divergence in the generation (at the expense of being slower) is still a useful property.

5.3 Hole Generation Analysis

A total of 211 point sets were used to run the analysis. The point sets were a subset of the same real-world test sets used with timing the random simple polygon generation. 500 random seeds were generated and those seeds were used to seed the random generator for each point set. A simple polygon was generated with the Continue variation, and the polygon was then used in the hole generation. For a single rotation run, we have a maximum amount of times we will try and rotate the point set to get a run that generates holes, up to 32 768, and the first successful run that returns holes within the maximum is recorded.

Figure 5.13 shows the timing result of a sweep with no rotation of the point set. The Y-axis is the time it took for the sweep line run, with the datapoints representing the maximum duration for each point set for each number of desired holes from one hole to max holes. It is for clarity that the other 499 runs for each number of desired holes for each pointset is omitted. "Max holes" was generated by asking for as many holes as there are points.

Figure 5.14 shows a datapoint for the run with the maximum duration for each point set, and

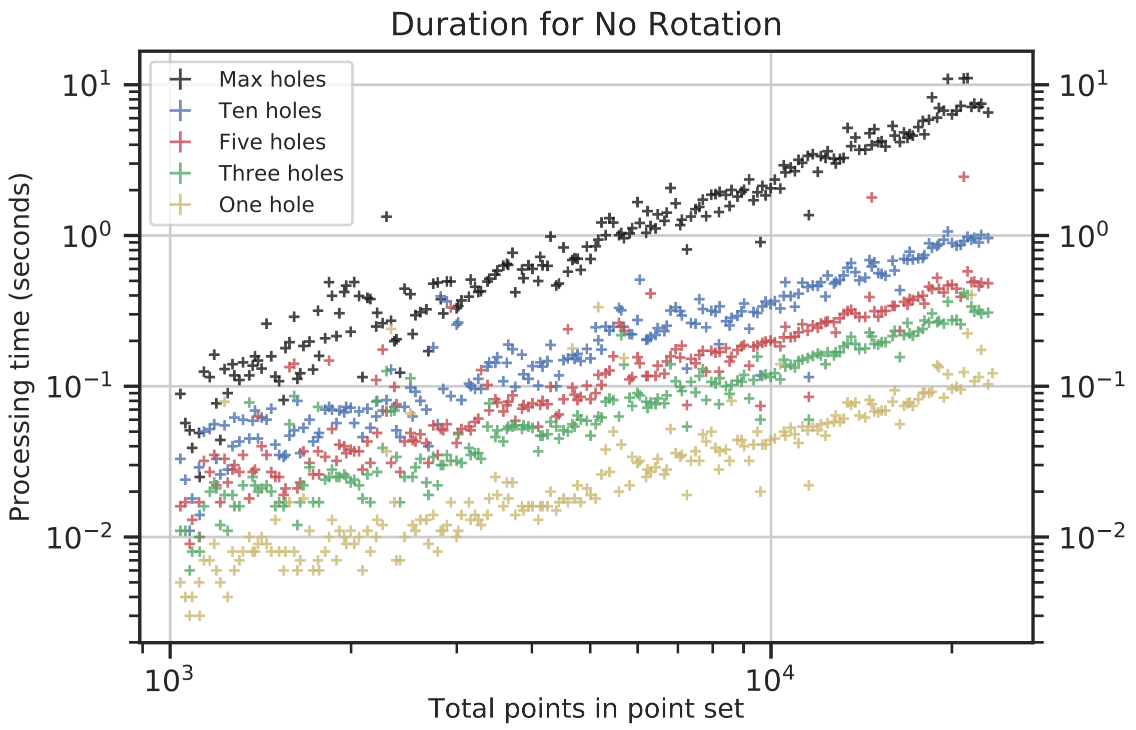

Figure 5.13: Timings without rotations.

for each number of desired holes, again the other 499 runs per holes per point set is omitted for clarity. The time duration represents the timing of the first run that returned a result.

In Figure 5.15 "No rotation" shows the maximum number of holes generated for each point set without a rotation, and "Rotation" shows the maximum number of holes generated after a single rotation. "Two sweeps" shows the maximum number of holes generated for each point set, where a "no rotation" run was done on top of the output of a "Rotation" run. The result is that we do get access to more possible holes if we rotate a set, even when asking for a maximum number of holes for each run.

Out of the rotations runs, 90.82% of the total runs only needed a single rotation to get a result with a hole, 1.18% of the total runs needed more than a single rotation to get a result, and 8.00% of the total runs did not get a result with a hole within a maximum number of attempted rotations. Figure 5.16 shows that the runs which either did not get a hole result, or needed more than one rotation are associated with certain specific problematic sets. The red dotted lines represent a point set that did not successfully return any holes after attempting up to the maximum required rotations, and the blue datapoints are the number of attempts to try and rotate until a successful run was generated. The figure shows that for a given simple polygon on a problematic set, a rotation might need more than one attempt to successfully rotate the point set to generate a hole, or in some cases it might not be able to generate holes with a rotation within a reasonable number of rotation attempts.

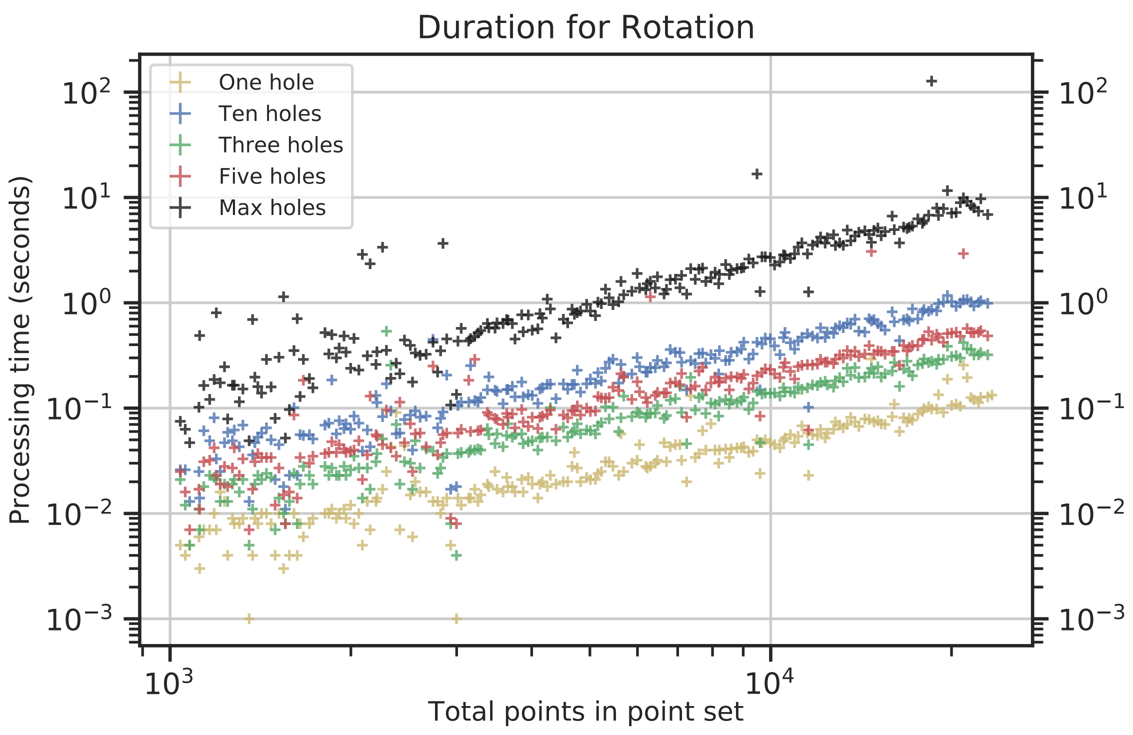

Figure 5.14: Timings with rotations.

No problems were discovered with the non-rotated version of the generator and we were able to generate holes with all the test sets in a single run.

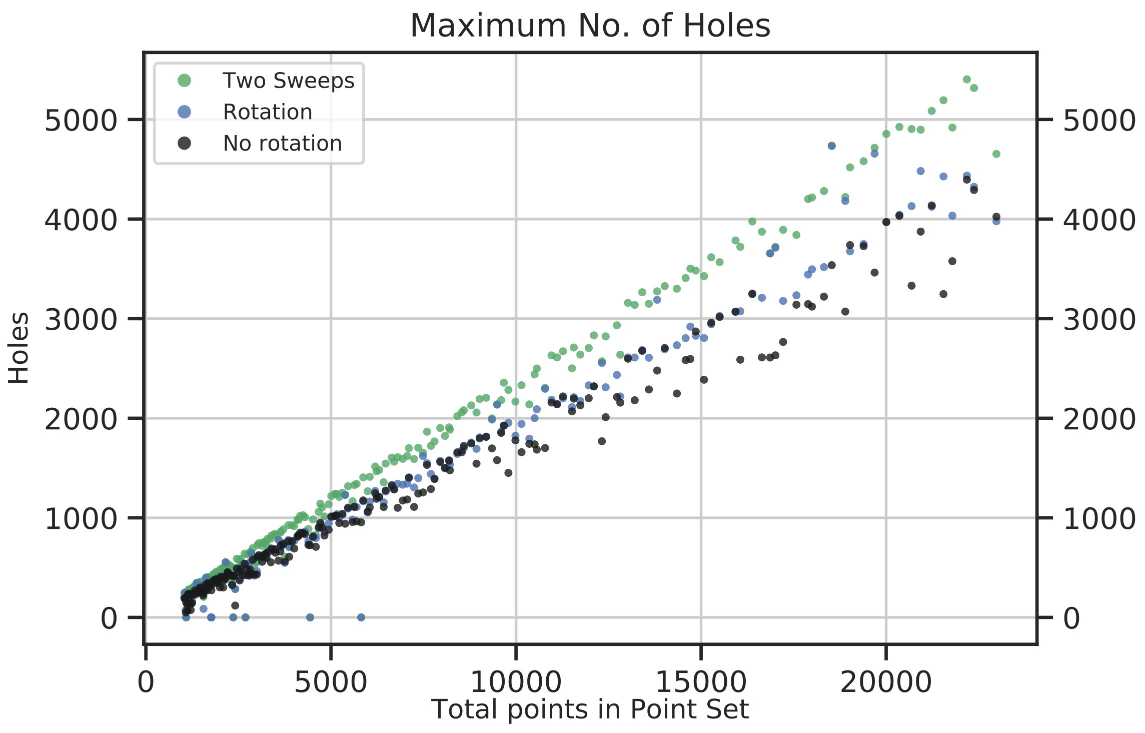

Figure 5.15: Maximum number of holes generated for real world data.

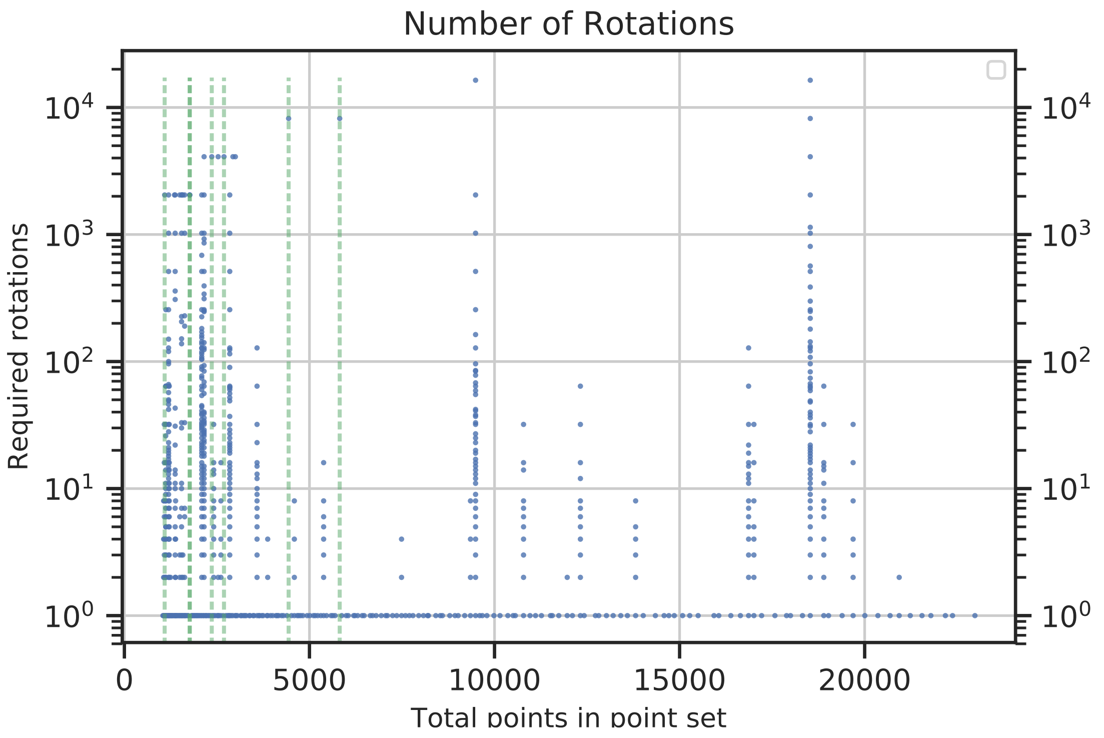

Figure 5.16: The number of attempts necessary to generate holes through rotations, with 90.82% of runs needing only a single rotation. The red dotted lines represent the position of sets that were unsuccessful in returning a result, and the blue dots represent the number of rotations attempted until a successful run generated holes.

CHAPTER 6

CONCLUSION

In this thesis we detailed variations in the heuristic of untangling intersections using the sweep line algorithm. Analysis shows that using a simpler model for the sweep line, and untangling an intersection as soon as it is discovered, increases the practical time efficiency compared to an older known heuristic. Looking at the average speed increase, the Reverse variation was the fastest at an average of 6 times faster, the Repeat Index was 4.5 faster, the Continue variation was 3 times faster, and finally with the Random variation being slowest of the practical variations at 1.2 times faster.

In addition, a criterion for analysing the random generation was developed that can detect a bias in the random distribution away from a uniform distribution of the generated simple polygons. The result is that there is a difference between the variations tested. The speedup of the Reverse variation affects the distribution of the generated simple polygons the most, which means the Reverse variation is recommended when speed is important, but when a good balance between a low bias in the randomness and speed is necessary, the Continue variation is recommended.

We also detailed a new heuristic to generate holes for a given modifiable simple polygon with modifiable holes that uses the sweep line algorithm as a basis. The heuristic has a time complexity of $\mathcal{O}(n \log n)$, and memory complexity of $\mathcal{O}(n)$ for a single sweep.

Future work for the simple polygon generation could be to create a binary search tree for the SLS that catches an intersection while inserting or removing an edge rather than just comparing the adjacent edges once an edge is inserted, it should be an effective way to increase the randomness of which intersection is untangled.

In regards to the hole generator, it would be of interest to find a good criterion to measure the randomness of the generated holes, as the distribution of the generated holes is possibly not connected to the randomness of finding suitable pair of edges.

The rotational version of the hole generator could also be improved, for instance by implementing a method for the rotation of a point set that would maintain collinearities when dealing with floating point values.

APPENDIX A

Algorithm 9 Line Segment $\prec$ Comparator

```
input: line segments: $e$ and $f$ (the cases in Chapter 2 in parenthesis below)
output: return true if $e \prec f$
**If** $e.p_1 = f.p_1$ and $e.p_2 = f.p_2$ **then**
    **return** false (c:0)
**If** $f.p_2 \leq e.p_1$ **then**
    **return** false (c:1)
**If** $e.p_2 \leq f.p1$ **then**
    **return** true (c:1)
**If** $e.p_1 = f.p_1$ **then**
    $usep_2$ = true (c:2)
// for collinear cases: $lo.p_{1/2}$ lex. lower than $hi.p_{1/2}$
$lo = e$; $hi = f$; $e_{lower}$ = true;
**If** $usep_2$ **then**
    **If** $f.p_2 < e.p_2$ **then**
        $lo = f$; $hi = e$;
        $e_{lower}$ = false;
**Else If** $f.p1 < e.p1$ **then**
    $lo = f$; $hi = e$;
    $e_{lower}$ = false;
**If** $usep_2$ **then**
    $val = lo$.det($hi.p_2$)
    $check =$ sign($val$) (c:2)
**Else**
    $val = lo$.det($hi.p1$)
    $check =$ sign($val$) (c:3,4,5)
**If** $check$ **then**
    **If** $e_{lower}$ **then**
        **return** false
    **Else**
        **return** true
**Else**
    **If** $e_{lower}$ **then**
        **return** true
    **Else**
        **return** False
```

A.1 Angle implementation for the hole algorithm

The angle code uses *atan2* from the standard math library of C++. Given a coordinate value (x, y), it returns an angle result from $(-\pi, \pi]$ which represents the angle between the vector from $(0, 0)$ to $(\infty, 0)$ and the vector from $(0, 0)$ to (x, y).

We assume that we have an edge e and a point p we want to get the angle to.

Sweeping left to right, the origin of the angle is always e_1, sweeping right to left, the origin is always e_2.

We define p_1 as the origin point of the angle, p_2 as the other endpoint of the line segment. Next we get the angles to p_2 and p from p_1, $angle_1 = atan2(p_2 - p_1)$ and $angle_2 = atan2(p - p_1)$.

There are two angles between the line segments $[p_1, p_2]$ and $[p_1, p]$, a convex angle and an concave angle. If $|angle_2 - angle_1|$ is larger than π then it is the concave angle, otherwise it is the convex angle. The angle $2\pi - |angle_2 - angle_1|$ is then the complement angle.

The side of e that is interior to the polygon determines whether we return the convex or the concave angle, see Figure A.1. We check which side is the interior side of e and use a determinant check to see which side the point p is on. If the two sides match, then we return the convex angle, otherwise we return the concave angle, see Algorithm 10.

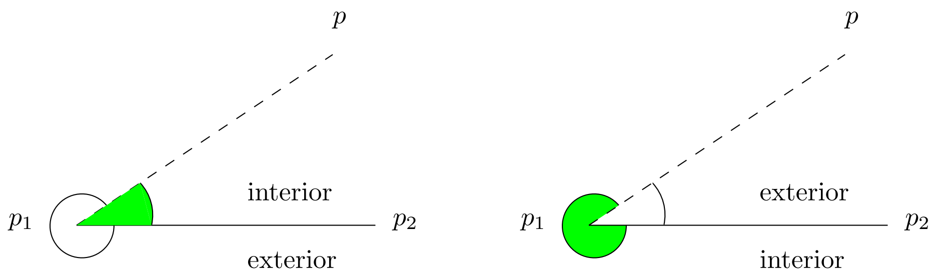

Figure A.1: The returned green angle depends on which side is the interior side of $[p_1, p_2]$.

Algorithm 10 Return Interior Angle function

```
1:  input: edge $e$, point $p$, boolean for origin of angle $usep_1$
2:  **If** $usep_1$ **then**
3:      $p_1 = e_1; p_2 = e_2$
4:  **Else**
5:      $p_1 = e_2; p_2 = e_1$
6:  $angle_1 =$ atan($p_2.y - p_1.y$, $p_2.x - p_1.x$)
7:  $angle_2 =$ atan($p.y - p_1.y$, $p.x - p_1.x$)
8:  $determinant =$ det($[p_1, p_2], p$)
9:  **If** $|angle_2 - angle_1 > \pi$ **then**
10:     $concave = |angle_2 - angle_1|$
11:     $convex = 2\pi - concave$
12: **Else**
13:     $convex = |angle_2 - angle_1|$
14:     $concave = 2\pi - convex$
15: **If** $usep_1$ **then**
16:     **If** interior side of $e$ is to the left **xor** $determinant < 0$ **then**
17:         return $convex$
18:     **Else**
19:         return $concave$
20: **Else**
21:     **If** interior side of $e$ is to the left **xor** $determinant < 0$ **then**
22:         return $concave$
23:     **Else**
24:         return $convex$
```

REFERENCES

[1] M. Sharir, A. Sheffer, and E. Welzl, “Counting plane graphs: Perfect matchings, spanning cycles, and Kasteleyn’s technique”, *Journal of Combinatorial Theory, Series A*, vol. 120, no. 4, pp. 777 –794, 2013. DOI: https://doi.org/10.1016/j.jcta.2013.01.002.

[2] A. Garcia, M. Noy, and J. Tejel, “Lower bounds on the number of crossing-free subgraphs of kn”, *Computational Geometry*, vol. 16, no. 4, pp. 211–221, 2000. DOI: https://doi.org/10.1016/S0925-7721(00)00010-9.

[3] Y. Nakahata, T. Horiyama, S. Minato, and K. Yamanaka, “Compiling crossing-free geometric graphs with connectivity constraint for fast enumeration, random sampling, and optimization”, *CoRR*, vol. abs/2001.08899, 2020. arXiv: 2001.08899.

[4] J. O’Rourke and M. Virmani, “Generating random polygons”, Department of Computer Science, Stoddard Hall, Smith College, Northampton, MA 01063, Tech. Rep. 011, Jul. 1991.

[5] T. Auer, “Heuristics for the generation of random polygons”, (Diploma thesis), University of Salzburg, Salzburg, Austria, Jun. 1996.

[6] P. Mayer, “Generating random simple polygons by vertex translations and insertions”, (Diploma thesis), University of Salzburg, Salzburg, Austria, Jul. 2020. DOI: 10.13140/RG.2.2.28734.92489.

[7] A. P. Tomás and A. L. Bajuelos, “Generating random orthogonal polygons”, in *Current Topics in Artificial Intelligence*, R. Conejo, M. Urretavizcaya, and J.-L. Pérez-de-la Cruz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 364–373, ISBN: 978-3-540-25945-9. DOI: http://dx.doi.org/10.1007/978-3-540-25945-9_36.

[8] A. P. Tomás and A. L. Bajuelos, “Quadratic-time linear-space algorithms for generating orthogonal polygons with a given number of vertices”, in *Computational Science and Its Applications – ICCSA 2004*, A. Laganá, M. L. Gavrilova, V. Kumar, Y. Mun, C. J. K. Tan, and O. Gervasi, Eds., Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2004, 117–126, ISBN: 978-3-540-24767-8. DOI: 10.1007/978-3-540-24767-8_13.

[9] R. Paul, A. Sarkar, and A. Biswas, “Construction of simple isothetic polygon from a set of points”, in *2018 Fifth International Conference on Emerging Applications of Information Technology (EAIT)*, (Kolkata, India), 2018, pp. 1–4, ISBN: 978-1-5386-3720-3. DOI: 10.1109/EAIT.2018.8470433.

[10] D. Dailey and D. Whitfield, “Constructing random polygons”, *SIGITE’08: Proceedings of the 9th ACM SIG-Information Technology Education Conference*, pp. 119–124, Jan. 2008. DOI: 10.1145/1414558.1414592.

[11] C. Zhu, G. Sundaram, J. Snoeyink, and J. S. B. Mitchell, “Generating random polygons with given vertices”, *Computational Geometry*, vol. 6, no. 5, pp. 277–290, 1996. DOI: 10.1016/0925-7721(95)00031-3.

[12] S. Sadhu, N. Kumar, and B. Kumar, “Random polygon generation through convex layers”, *Procedia Technology*, vol. 10, pp. 356–364, 2013, First International Conference on Computational Intelligence: Modeling Techniques and Applications (CIMTA) 2013. DOI: https://doi.org/10.1016/j.protcy.2013.12.371.

[13] C. Sohler, “Generating random star-shaped polygons”, in *Proceedings of the 11th Canadian Conference on Computational Geometry, UBC, Vancouver, British Columbia, Canada, August 15-18*, 1999.

[14] J. A. Shufelt and H. J. Berliner, “Generating hamiltonian circuits without backtracking from errors”, *Theoretical Computer Science*, vol. 132, no. 1, pp. 347 –375, 1994. DOI: https://doi.org/10.1016/0304-3975(94)90239-9.

[15] H. Jiang, *A new constraint of the hamilton cycle algorithm*, 2017. arXiv: 1710.06974 [cs.DM].

[16] J. B. Robinson, *On the hamiltonian game (a traveling-salesman problem).* Santa Monica, CA: RAND Corporation, 1949.

[17] M. M. Flood, “The traveling-salesman problem”, *Operations Research*, vol. 4, no. 1, 61–75, 1956. DOI: 10.1287/opre.4.1.61.

[18] G. A. Croes, “A method for solving traveling-salesman problems”, *Operations Research*, vol. 6, no. 6, 791–812, Dec. 1958.

[19] M. I. Shamos and D. Hoey, “Geometric intersection problems”, *17th Annual Symposium on Foundations of Computer Science (FOCS 1976)*, pp. 208–215, 1976. DOI: 10.1109/SFCS.1976.16.

[20] J. L. Bentley and T. A. Ottmann, “Algorithms for Reporting and Counting Geometric Intersections”, *IEEE Trans. Comput.*, vol. 28, no. 9, 643–647, Sep. 1979. DOI: 10.1109/TC.1979.1675432.

[21] J. van Leeuwen and A. A. Schoone, “Untangling a travelling salesman tour in the plane”, in *Proceedings of the 7th Conference Graphtheoretic Concepts in Computer Science (WG ’81), Linz, Austria, June 15-17, 1981*, J. R. Mühlbacher, Ed., Hanser, Munich, 1981, pp. 87–98.

[22] J. J. Bentley, “Fast algorithms for geometric traveling salesman problems”, *ORSA Journal on Computing*, vol. 4, no. 4, Dec. 1992. DOI: 10.1287/ijoc.4.4.387.

[23] S. Hougardy, F. Zaiser, and X. Zhong, *The approximation ratio of the 2-opt heuristic for the metric traveling salesman problem*, 2020. arXiv: `1909.12025 [cs.DM]`.

[24] J. R. Shewchuk, “Adaptive precision floating-point arithmetic and fast robust geometric predicates”, *Discrete & Computational Geometry 18(3)*, vol. 18, 305–363, Oct. 1997. DOI: `10.1007/PL00009321`.

[25] D. E. Knuth, *The art of computer programming, volume 2 (3rd ed.): Seminumerical algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc., 1997, vol. 2, pp. 139–140, ISBN: 0201896842. DOI: `10.5555/270146`.